Fall 9-1-1995

Improving the Performance of Conservative Generational Garbage Collection ; CU-CS-784-95

David Barrett
University of Colorado Boulder

Improving the Performance of Conservative Generational Garbage Collection

David A. Barrett

CU-CS-784-95 September 1995

University of Colorado at Boulder

Technical Report CU-CS-784-95
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Improving the Performance of Conservative Generational Garbage Collection

by

David A. Barrett

B.S. University of Virginia, 1981
M.S. University of Colorado, Boulder, 1990

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
1995

This thesis for the Doctor of Philosophy degree by
David A. Barrett
has been approved for the
Department of
Computer Science
by

Benjamin G. Zorn

Dirk Grunwald

Date

Barrett, David A. (Ph.D., Computer Science)

Improving the Performance of Conservative
Generational Garbage Collection

Thesis directed by Dr. Benjamin G. Zorn

Abstract

As object-oriented features are added to programming languages such as C++, demands upon dynamic memory allocation systems increase. Prototyping languages such as Smalltalk and LISP have long recognized the impact of garbage collection on improving reliability and programmer productivity, but providing garbage collection in C++ is hard because pointers are difficult to reliably identify. Conservative generational collectors provide a partial solution to this problem at the cost of increased memory consumption due to memory fragmentation and tenured garbage.

This research introduces a new conservative generational garbage collection mechanism, the *Dynamic Threatening Boundary (DTB)*, that, unlike previous collectors, uses a dynamically updated threatening boundary to select objects eligible for reclamation. By using object lifetime demographics to adjust the threatening boundary between generations backward during run-time, one policy using the DTB mechanism significantly reduced tenured garbage. Because this mechanism clearly separates generation selection policy from implementation, future researchers can easily develop other new policies that may reduce costs further.

Existing implementations of generational collectors for C often use an expensive virtual memory write-trap to maintain pointers into the scavenged generation. This research investigates the costs and benefits of using an explicit store check instruction sequence to reduce the time overhead of the write barrier for generational collectors for C. To further reduce memory fragmentation and time overhead, this research also shows new ways to predict short-lived objects and recognize long-lived ones.

The effectiveness of these techniques is evaluated by using trace-driven simulation to compare them against several existing generational collection policies. Although program behavior strongly influences the costs of garbage collection, relevant C program behaviors have not been measured and reported before. Several well-known C programs were instrumented and the results analyzed to provide data for future researchers incorporating garbage collection into languages that currently do not provide it.

Acknowledgements

I gratefully acknowledge the help of so many people whom have made this dissertation possible. I thank my advisor, Dr. Benjamin Zorn, whose moral, economic, and technical support was invaluable to the completion of this work. His tireless attention to detail and quality feedback have been of immeasurable help. Next, I would like to thank Martha Escobar and Richard Wolniewicz who both took time from their own jobs and dissertations work to read this entire document and suggest necessary improvements. I also wish to thank Barbara Ryder and the volunteers of the ACM SIGPLAN committees who provided financial assistance allowing me to present this material at the Programming Language Design and Implementation (PLDI) conferences in 1993 and 1995. I also thank my thesis committee: Wayne Citrin, Dirk Grunwald, Clayton Lewis, Rod Oldehoeft, Mike Schwartz, and William Waite for providing additional comments and suggestions for improvements to this work.

Contents

Tables

Figures

Chapter 1

Introduction

1.1 Motivation

Since the beginning of this decade, industrial software systems have undergone significant change. Drastically lower hardware costs have correspondingly increased the number and power of computers that modern society uses. The demands upon these machines have grown too as people have discovered more ways to put the increased computational resources to use. The result is that the size and number of software systems have grown proportionally, forcing changes in the way people develop them.

Larger software systems and lower-cost hardware have caused more powerful programming languages to become popular in industry. These languages provide cleaner mechanisms for data abstraction, which in turn, require more sophisticated run-time systems to support them. For example, event-driven programming, which used to be almost strictly within the domain of operating systems device drivers, has now migrated upward into the typical application programmer's domain as illustrated by common use of window-based user interfaces. Window systems frequently use an object-oriented paradigm, and that paradigm encourages heavy dynamic storage allocation.

These complex applications, with their sophisticated data structures, place heavy demands upon the computer's primary memory. In order to use this memory more effectively, programs are allocating and deallocating memory as the program runs rather than relying primarily upon static storage allocation methods. C++, with its object-oriented style, encourages programmers to create objects that have lifetimes quite different than the scope of their names. This increased use of dynamic memory allocation also requires more powerful run-time support from the memory systems than has been typical in widespread industrial use in the past.

Garbage collection frees the programmer and designer from the burden of memory resource management by letting the computer determine when storage may be reused. Program design is simplified because data structures using objects shared between modules do not require a protocol for freeing their storage. Reliability and development time are improved because errors in memory management often cause catastrophic failure and are time consuming to test, locate, and repair.

The benefits of garbage collection come at a cost, however. Garbage collector implementations typically require more memory than explicit deallocation. The virtual memory system may thrash when a garbage collector attempts to identify unused memory (garbage). Processor overhead can also be excessive and disruptive if the collector used is not adequately matched to the application. Finally, programming languages and computer architecture strongly influence garbage collector design, often by making collection difficult to implement efficiently.

Past research has produced two compatible techniques of garbage collection that partially address these problems: generational and conservative collection.

Generational collectors work by dividing allocated objects into young and old generations based upon their age. Generational collectors examine older objects less frequently than younger ones. To avoid erroneously deallocating an object, generational collectors use a *write barrier* to record pointers from old objects to young ones by checking every store instruction in the program. Generational garbage collectors reduce thrashing and disruptive

collection pauses by limiting their scope only to objects that have been recently allocated. Empirical measurements show that such objects are often more likely to be available for reclamation than older ones [BZ93, Hay91, WM89b, ZG92].

Conservative collectors never deallocate objects referenced by values that only appear to be pointers and never move allocated objects. Thus, conservative collectors can be used with programming languages and computer architectures that make distinguishing pointers from data difficult (e.g., C++, RISC). Since conservative collectors do not move live objects, they do not have to reliably recognize and alter pointers to them.

The success of both collection algorithms is evinced by their frequent use in language environments that require automatic storage reclamation [App92, BDS91, Fra92, HMS92].

Despite their success, both types of collectors have drawbacks. Generational collectors create *tenured garbage* as old objects become unreachable and are not collected [BZ95]. Store checks done by the write barrier can cause high processor overhead [HMS92, Zor90a]. Conservative collectors can cause memory fragmentation since live objects cannot be copied and they can become scattered across the address space.

1.2 Research Contributions

This research investigates ways to improve the performance of conservative generational garbage collection. First, I determine the impact of using an in-line write barrier for generational collection upon processor overhead. Second, I measure C program behaviors relevant to garbage collection, and how they affect performance. Third, I evaluate a new method of generational garbage collection to reduce tenured garbage. This method allows explicit separation of garbage collection policy from implementation. Fourth, I show how memory fragmentation may be reduced by using information available at the time of allocation to predict the lifetime of short-lived objects.

The write barrier can be a source of significant processor overhead in generational garbage collectors. When pointers can be recognized with certainty, as is possible with LISP and Smalltalk, the write barrier is frequently implemented by instructions added at each store instruction. The instruction sequence checks for creation of a pointer from an uncollectable old object to a collectable young object.

When pointers cannot be identified, as is common with C++ and C, conservative generational collectors typically use an operating system call to write protect memory containing old objects. Instead of using instructions at each store, a write-protect trap causes the operating system to call a subroutine when a store instruction tries to write a value into the protected memory. The subroutine then records the stored location for future reference by the collector and returns. These traps can be very expensive depending upon the operating-system implementation and program behavior.

This research investigates the new approach of using an in-line write barrier for conservative generational garbage collection. Such an approach becomes practical with the increased availability of binary-to-binary translators, which allow executable programs to be altered without requiring compiler modification. Trap handling in operating systems is often very slow, so replacing this overhead with a few instructions is a promising approach.

The popularity of object-oriented programming has caused new features to be added to programming languages that do not have garbage collection. Object-oriented features require lots of dynamic storage allocation, which makes the addition of garbage collection attractive. Program behavior strongly affects garbage collection. The relevant behaviors of programs written in languages that do not provide garbage collection have not been measured before.

This research measures aspects of C program behavior relevant to conservative generational garbage collection. Specifically, the store frequency patterns, which have never been measured before for C, are evaluated in the context of their impact upon an in-line write barrier. Object life time demographics, which are relevant to generation selection, are also measured.

Generational collectors collect older generations less frequently than younger ones. Pause times are reduced but at the expense of maintaining the write barrier and retaining tenured garbage. Tenured garbage is created when the generation boundaries do not match the object demographics. For example, if the average object age changes during the lifetime of the program, then

no fixed-age generation boundary policy will work well. Either processor overhead will be excessive if the youngest generation is too big, or tenured garbage will be created if it is too small.

This research investigates a new generational partitioning mechanism that allows the boundary between the old and young generations to be adjusted dynamically. Also, a specific policy using this mechanism to partition objects based upon their lifetime demographics is evaluated. The processor time and memory overhead of this approach is compared against several existing widely-used generational algorithms using several well-known C applications. By providing an approach that clearly separates policy from implementation, the door is opened for future rearchers to investigate other policies as well.

Conservative collectors can cause memory fragmentation because live objects may not be moved after they are allocated. Moving objects requires pointers to be altered to reflect each object's new location. Altering a value that was misidentified as a pointer by a conservative collector would break the program. As young objects die, the resulting gaps in the address space become fragmented by older objects. To address this problem a reliable way is needed to predict the lifetimes of objects at the time they are allocated.

This research investigates new ways to detect when objects die and thus reduce memory consumption. One approach shows how short-lived objects can be reliably predicted by their *allocation site*—what point in the program did the allocation. Segregating such objects reduces fragmentation. Another approach recognizes long-lived clusters of objects by using the write barrier to monitor pointer stores to collect them less often, thereby saving processor overhead. Lastly, store behavior discovered by the write barrier may be used to indicate the death of object clusters.

The remainder of this dissertation proceeds as follows. First, the relationship to prior research is discussed. Next, the implementation model appropriate to garbage collection is presented with the corresponding costs. Next, the experimental methods and assumptions used to evaluate this work are presented. After that, a chapter appears for each of the principal contributions: the in-line write barrier, the dynamic threatening boundary, and lifetime prediction. Lastly, the conclusion will discuss how these results relate to each other and to future work.

Terms commonly used in the literature will appear frequently throughout this document. In order to assist the reader, such terms will be italicized when they are first used and will appear both in the glossary on page 53 and in the index.

Chapter 2

Related Work and Contributions

This chapter presents the context required for understanding and evaluating the remainder of this dissertation. First, problems of explicit storage allocation motivate the need for garbage collection. Next, general concepts of garbage collection are presented along with how two types of collection, generational and conservative, reduce the costs and expand the availability of automatic storage reclamation. Lastly, advantages and limitations of specific examples of the prior art and how they inspired the research done for this dissertation are discussed.

2.1 Problems with Explicit Deallocation

Allocation of storage for program data structures is one of the most important services provided by the programming environment. Static allocation provides an efficient and simple solution for objects such as global variables (e.g., FORTRAN, BASIC) where object sizes are fixed, known in advance, and lifetimes are the same as the program's. Stack allocation provides a simple way to reuse memory when the object lifetimes match the scope of a subroutine (e.g., local variables in PASCAL or automatic variables in C). When lifetimes or object sizes are not known in advance, such as for recursive data structures or for dynamic arrays and records, dynamic storage allocation is required.

Several forms of dynamic storage allocation exist. In the simplest form, storage is allocated explicitly by the program and is never deallocated: the operating system recovers the memory when the program terminates. For short-lived applications or on machines with large amounts of virtual memory, such an approach is often appropriate; it has the advantages of minimal programmer effort and low probability of storage deallocation errors. But, when storage demands are large over the lifetime of the program, memory must somehow be reused.

One common approach requires the programmer to explicitly deallocate each allocated *object* (piece of storage) at the end of its lifetime (e.g., C, ADA). Unfortunately, determining when deallocation should occur is often difficult and becomes rapidly more so as the number of modules sharing a given data structure grows.

Diagnosis and repair of incorrect deallocations is notoriously difficult and expensive. Forgetting to deallocate an object results in a *memory leak* in which storage not reclaimed grows slowly until memory is exhausted; identifying which objects cause such leaks is very difficult. Conversely, deallocating an object that is later referenced creates a *dangling reference* that will not be apparent until the storage reclaimed from that object is overwritten and a reference to the original object occurs. In both cases, the failure symptom often occurs long after the cause. Tools such as Purify [HJ92] (a sort of memory debugger) help diagnose these problems, but only after they occur and with manual intervention.

Programmers must still design and strictly follow a protocol for the freeing of objects. This protocol can become especially cumbersome when object references may be *shared* between modules and written by different engineers or companies. The protocol for one module may have been written without knowledge of protocol for the other and may conflict with each other.

When these two problems are likely to occur in a software system, automatic storage deallocation

becomes attractive.

2.2 Garbage Collection Algorithms

Garbage collection provides a solution where storage reclamation is automatic. This section provides an overview of the the simplest approaches to garbage collection, and then discusses the two forms of garbage collection most relevant to this dissertation: generational collection and conservative collection.

2.2.1 Simple Approaches

All garbage collection algorithms attempt to deallocate objects that will never be used again. Since they cannot predict future accesses to objects, collectors make the simplifying assumption that any object that is accessible to the program will indeed be accessed and thus cannot be deallocated. Thus, garbage collectors, in all their variety, always perform two operations: identify *unreachable objects* (*garbage*) and then deallocate (*collect*) them.

Reference-counting collectors identify unreachable objects and deallocate them as soon as they are no longer referenced [Col60, Knu73]. Associated with each object is a reference count that is incremented each time a new *pointer* to the object is created and decremented each time one is destroyed. When the count falls to zero, the reference counts for immediate descendents are decremented and the object is deallocated. Unfortunately, reference-counting collectors are expensive because the counts must be maintained and it is difficult to reclaim circular data structures using only local reachability information.

Mark-sweep collectors are able to reclaim circular structures by determining information about global reachability [Knu73, McC60]. Periodically, (e.g., when a memory threshold is exhausted) the collector marks all reachable objects, and then reclaims the space used by the unmarked ones. Mark-sweep collectors are also expensive because every dynamically-allocated object must be visited; the *live* ones during the mark phase and the dead ones during the sweep phase.

On systems with virtual memory where the program address space is larger than primary memory, visiting all these objects may require the entire contents of dynamic memory be brought into primary memory each time a collection is performed. Also, after many collections, objects become scattered across the address space because the space reclaimed from unreachable objects is fragmented into many pieces by the remaining live objects. Explicit deallocation also suffers from this problem. Scattering reduces reference locality and ultimately increases the size of primary memory required to support a given application program.

Copying collectors provide a partial solution to this problem [Bak78, Coh81]. These algorithms mark objects by copying them to a separate contiguous area of primary memory. Once all the reachable objects have been copied, the entire address space consumed by the remaining unreachable objects is reclaimed at once; garbage objects need not be swept individually. Because in most cases the ratio of live to dead objects tends to be small (by selecting an appropriate collection interval), the cost of copying live objects is more than offset by the drastically reduced cost of reclaiming the dead ones. As an additional benefit, spatial locality is improved as the copying phase compacts all the live objects. Finally, allocation of new objects from the contiguous free space becomes extremely inexpensive. A pointer to the beginning of the free space is maintained; allocation consists of returning the pointer and incrementing it by the size of the allocated object.

But copying collectors are not a panacea; they cause disruptive pauses and they can only be used when pointers can be reliably identified. Long pauses occur when a large number of reachable objects must be traced at each collection. *Generational* collectors reduce tracing costs by limiting the number of objects traced [LH83, Moo84, Ung84].

Precise run-time–type information available for languages such as LISP, ML, Modula-3 and Smalltalk allows pointers to be reliably identified. However, for languages such as C or C++, copying collection is difficult to implement because lack of run-time type information prevents pointer identification. One solution is to have the compiler provide the necessary information [DMH92]. *Conservative* collectors provide a solution when such compiler support is unavailable [BW88].

2.2.2 Generational Collection

For best performance, a collector should minimize the number of times each reachable object is traced during its lifetime. *Generational collectors* exploit the experimental observation that old objects are less likely to die than young ones by tracing old objects less frequently [BZ93, Bak93, Hay91, Sha87, SM94, WM89b, Zor89, ZG92]. Since most of the dead objects will be young, only a small fraction of the reclaimable space will remain unreclaimed after each collection, and the cost of frequently retracing all the old objects is saved. Eventually, even the old objects will have to be traced to reclaim long-lived dead objects. Generational collectors divide the memory space into several *generations* where each successive older generation is traced less frequently than the younger generations. Adding generations to a copying collector reduces scavenge time pauses because old objects are neither copied nor traced on every collection.

Generational collectors can avoid tracing objects in the older generation when pointers from older objects to younger objects are rare. Tracing the old objects is especially expensive when they are in paged-out virtual memory on disc. This cost increases as the older generations become significantly larger than younger ones, as is typically the case. One way implementations of generational collectors reduce tracing costs is to segregate large objects that are known not to contain pointers are into a special untraced area [UJ92]. Another way to reduce costs is to maintain forward-in-time inter-generational pointers explicitly in a collector data structure, the *remembered set*, which becomes an extension of the root set. When a pointer to a young object is stored into an object in an older generation, that pointer is added into the remembered set for the younger generation. Tracking such stores is called maintaining the *write barrier*. Stores from young objects to old ones are not explicitly tracked. Instead, whenever a given generation is collected, all younger generations are also collected.

The write barrier is often maintained by using virtual memory to write protect pages that are eligible to contain such pointers [AEL88]. Another method is to use explicit in-line code to check for such stores. Such a check may be implemented by the compiler, but other approaches are possible. For example, a post processing program may be able to recognize pointer stores in the compiler output, and insert the appropriate instructions [Dig91, HJ92, LB94, SE94].

Designers of generational collectors must also establish the size, collection and promotion policies for each generation and how many generations are appropriate. The collection policy determines when to collect; the number of generations, their size, and the promotion policy determines what is collected.

The collector must determine how frequently to scavenge each generation; more frequent collections reduce memory requirements at the expense of increased CPU time because space is reclaimed sooner but live objects are traced more frequently. As objects age, they must be *promoted* to older generations to reduce scavenge costs; promoting a short-lived object too soon may cause space to be wasted because it may be reclaimed long after it becomes unreachable; promoting a long-lived object too late results in wasted CPU time as that object is traced repeatedly. The space required by each generation is strongly influenced by the promotion and scavenge policies.

If the promotion policy of a generational collector is chosen poorly, then *tenured garbage* will cause excessive memory consumption. Tenured garbage occurs when many objects that are promoted to older generations die long before the generation is scavenged. This problem is most acute with a fixed-age policy that promotes objects after a fixed number of collections. Ungar and Jackson devised a policy that uses object demographics to delay promotion of objects until the collector's scavenge costs require it [UJ92].

Because generational collectors trade CPU time maintaining the remembered sets for a reduced scavenge time, their success depends upon many aspects of program behavior. If objects in older generations consume lots of storage, their lifetimes are always long, they contain few pointers to young objects, pointer stores into them are rare, and many objects die at a far younger age, then generational collectors will be very effective. However, even generational collectors must still occasionally do a full collection, which can cause long delays for some programs. Often, however, collectors provide tuning mechanisms that must be manipulated directly by the end user

to optimize performance for each of their programs [App92, Sym85, Xer83].

Generational collectors have been implemented successfully in prototyping languages, such as LISP, Modula-3, Smalltalk and PCedar [App92, BDS91, Cou88, HMS92, Boe94, Nel91, Sym85]. These languages share the characteristic that pointers to objects are readily identifiable, or hardware tags are used to identify pointers. When pointers cannot be identified, copying collectors cannot be used, for when an object is copied, all pointers referring to it must be changed to reflect its new address. If a pointer cannot be distinguished from other data then its value cannot be updated because doing so may alter the value of a variable. The existing practice in languages such as C and C++, which prevent reliable pointer identification, has motivated research into *conservative non-copying* collectors.

2.2.3 Conservative Collection

Conservative collectors may be used in language systems where pointers cannot be reliably identified [BW88]. Indeed, an implementation already exists that allows a C programmer to retrofit a conservative garbage collector to an existing application [Boe94]. This class of collectors makes use of the surprising fact that values that look like pointers (*ambiguous* pointers) usually are pointers. Misidentified pointers result in some objects being treated as live when, in fact, they are garbage. Although some applications can exhibit severe leakage [Boe93, Wen90], usually only a small percentage of memory is lost because of conservative pointer identification.

Imprecise pointer identification causes two problems: valid pointers to allocated objects may not be recognized (*derived* pointers), or non-pointers may be misidentified as pointers (*false* pointers). Both cases turn out to be critical concerns for collector implementors.

A *derived* pointer is one that does not contain the base address of the object to which it refers. Such pointers are typically created by optimizations made either by a programmer or a compiler and occur in two forms. *Interior* pointers are ones that point into the middle of an object. Array indices, and fields of a record are common examples [BGS94]. Sometimes an object that has no pointer into it from anywhere is still reachable. For example, an array whose lowest index is a non-zero integer may only be reachable from a pointer referring to index zero. Here the problem is that a garbage collector may mistakenly identify an object as unreachable because no explicit pointers to it exist.

With the exception of interior pointers, which are more expensive to trace, compiler support is required to solve this problem no matter what collection algorithm is used. In practice, it turns out that compiler optimizations have not been a problem yet (June 1995) because enabling sophisticated optimizations often breaks other code in the users program and is not used with garbage collected programs in practice [Boe95b]. Such support has been studied by other researchers and will not be discussed further in this dissertation [Boe91, DMH92, ED94].

False pointers exist when the type (whether it is a pointer or not) of an object is not available to the collector. For example, if the value contained in an integer variable corresponds to the address of an allocated but unreachable object, a conservative collector will not deallocate that object. A heuristic called *blacklisting* reduces this problem by not allocating new objects from memory that corresponded to previously discovered false pointers [Boe93]. But even when the type is available, false pointers may still exist. For example, a pointer may be stored into a compiler generated temporary (in a register or on the stack) that is not overwritten until long after its last use. While memory leakage caused by the degree of conservativism chosen for a particular collector is still an area of active research, it will not be discussed further in this dissertation except in the context of costs incurred by the conservative collector's pointer-finding heuristic.

Not only can false pointers cause memory leakage, but they also preclude copying. When a copying collector finds a reachable object, it creates a new one, copies the contents of the old object into it, deletes the original object, and overwrites all pointers to the old object with the address of the new object. If the overwritten pointer was not a pointer, but instead was the value of a variable, this false pointer cannot be altered by the collector. This problem can be partly solved by moving only objects that are not referenced through false pointers as in Bartlett's *Mostly Copying* collection algorithm [Bar90].

If true pointers cannot be recognized, then the collector may not copy any objects after they are created. One of the chief advantages of copying collectors, reference locality, is lost [Moo84]. A conservative collector can also cause a substantial increase in the size of a process's working set as long-lived objects become scattered over a large number of pages.

Memory becomes fragmented as the storage freed from dead objects of varying sizes become interspersed with long-lived live ones. This problem is no different than the one faced by traditional explicit memory allocation systems such as *malloc/free* in widespread use in the C and C++ community. Solutions to this problem may be readily transferable between garbage collection and explicit memory allocation algorithms.

The trace or sweep phases of garbage collection, which are not present in explicit memory allocation systems, can dramatically alter the paging behavior of a program. Implementations of copying collectors already adjust the order in which reachable objects are traced during the mark phase to minimize the number of times each page must be brought into main memory [DWH+90]. Zorn [Zor90b] has shown that isolating the mark bits from the objects in a *Mark-Sweep* collector and other improvements also reduce collector-induced paging. Generational collectors also dramatically reduce the pages referenced as well [DWH+90, Moo84, Ung84].

Even though generational collectors reduce pause times, work is also being done to make garbage collection suitable for the strict deadlines of real-time computing. Baker [Bak78] suggested *incremental collection*, which interleaves collection with the allocating program (*mutator*) rather than stopping it for the entire duration of the collection. Each time an object is allocated, the collector does enough work to ensure the current collection completes before another one is required.

Incremental collectors must ensure that traced objects (those that have already been scanned for pointers) are not altered, for if a pointer to an otherwise unreachable object is stored into the previously scanned object, that pointer will never be discovered and the object, which is now reachable, will be erroneously reclaimed. Although originally maintained by a read barrier [AEL88, Bak78] this invariant may also be maintained by a write barrier [BDS91]. The write barrier detects when a pointer to an untraced object is stored into a traced one, which is then retraced [Wil95b]. Notice that this barrier may be implemented by the same method as the one for the remembered set in generational collectors [DWH+90]; only the set of objects monitored by the barrier changes. Nettles and O'Toole [NO93] relaxed this invariant in a copying collector by using the write barrier to monitor stores into threatened objects and altering their copies before deallocation.

Because incremental collectors are often used where performance is critical [App92, BDS91, NO93], any technology to improve write barrier performance is important to these collectors. Conversely, high-performance collection of any type is more widely useful if designed so it may be easily adapted to become incremental. This dissertation will not explicitly discuss incremental collection further, but keep in mind that write-barrier performance applies to incremental as well as generational collectors.

2.3 Related Work

This dissertation combines and expands upon the work done by several key researchers. Xerox PARC developed a formal model and the concept of explicit threatened and immune sets. Ungar and Jackson developed a dynamic promotion policy. Hosking, Moss and Stefanović compared the performance of various write barriers for precise collection, and Zorn showed that in-line write barriers can be quite efficient. I shall now describe each of these works and then introduce the key contributions this dissertation will make and how they relate to the previous work.

2.3.1 Theoretical Models and Implementations

Researchers at Xerox PARC have developed a powerful formal model for describing the parameter spaces for collectors that are both generational and conservative [DWH+90, Hay90]. A garbage collection becomes a mapping from one storage state to another. They show that storage states may be partitioned into *threatened* and *immune* sets. The method of selecting these sets induces a specific garbage collection algorithm. A pointer augmentation provides the formalism for model-

ing remembered sets and imprecise pointer identifications. Finally, they show how the formalism may be used to combine any generational algorithm with a conservative one.

They used the model to design and then implement two different conservative-generational garbage collectors. Their *Sticky Mark Bit* collector uses two generations and promotes objects surviving a single collection. A refinement of this collector (*Collector II*) allows objects allocated beyond an arbitrary point in the past to be immune from collection and tracing. This boundary between old objects, which are immune, and the new objects, which are threatened, is called the *threatening boundary*. More recently, these authors have received a software patent covering their ideas [WDBH94].

Until now, Collector II was the only collector that made the threatening boundary an explicit part of the algorithm. It used a fixed threatening boundary and time scale that advanced only one unit per collection. This choice was made to allow an easy comparison with a non-generational collector, not to show the full capability of such an idea.

Both collectors show that the use of two generations substantially reduces the number of pages referenced by the collector during each collection. However, these collectors exhibited very high CPU overhead; the generational collectors frequently doubled the total CPU time.

In later work, they implemented a *Mostly Parallel* concurrent two-generation conservative *Sticky Mark Bit* collector for the PCeder language [BDS91]. This combination substantially reduced pause times for collection compared to a simple full-sweep collector for the two programs they measured. These collectors used page-protection traps to maintain the write barrier. They did so by write protecting the entire *heap* address space and installing a trap handler to update a dirty bit for the first write to each page. Pause times were reduced by conducting the trace in parallel with the mutator. Once the trace was complete, they stopped the mutator, and retraced objects on all pages that were flagged as dirty.

All their collectors shared the limitation that once promoted to the next generation, objects were only reclaimed when a full collection occurred, so scavenger updates to the remembered set were not addressed. Tenured garbage could only be reclaimed by collecting the entire heap.

My work extends upon theirs by exploiting the full power of their model to dynamically update the threatening boundary at each collection rather than relying only upon a simple fixed-age or full-collection policy.

2.3.2 Feedback-Mediation

Ungar and Jackson measured the effect of a dynamic promotion policy, *Feedback Mediation*, upon the amount of tenured garbage and pause times for four six-hour Smalltalk sessions [UJ92]. They observed that object lifetime distributions are irregular and that object lifetime demographics can change during execution of the program. This behavior affects a fixed-age tenuring policy by causing long pause times when a preponderance of young objects causes too little tenuring, and excessive garbage when old objects cause too much tenuring.

They attempted to solve this problem using two different approaches. First, they placed pointer-free objects (bitmaps and strings) larger than one kilobyte into a separate area; this approach was effective because such objects need not be traced and are expensive to trace and copy. Second, they devised a dynamic tenuring policy that used feedback mediation and demographic information to alter the promotion policy so as to limit pause times. Rather than promoting objects after a fixed number of collections, Feedback mediation only promoted objects when a pause-time constraint was exceeded because a high percentage of data survived a scavenge and would be costly to trace again. To determine how much to promote, they maintained object demographic information as a table containing of the number of bytes surviving at each age (where age is number of scavenges). The tenuring threshold was then set so the next scavenge would likely promote the number of bytes necessary to reduce the size of the youngest generation to the desired value.

Their collector appears similar to Collector II in that it uses an explicit threatening boundary, but differs because it does so for promotion only, not for selecting the immune set directly.

My work extends theirs by allowing objects to be demoted. Their object promotion policies can

be modeled by advancing the threatening boundary by an amount determined by the demographic information each time the pause-time constraint is exceeded. I extend this policy by moving the threatening boundary backward in time to reclaim the tenured garbage that was previously promoted.

Hanson implemented a movable threatening boundary for a garbage collector for the SNOBOL-4 programming language [Han77]. After each collection, surviving objects were moved to the beginning of the allocated space and the remaining (now contiguous) space was freed. Allocation subsequently proceeded in sequential address order from the free space. After the mark phase, and before the sweep phase, the new threatening boundary was set to the address of the lowest unmarked object found by a sequential scan of memory. This action corresponds to a policy of setting the threatening boundary to the age of the oldest unmarked object before each sweep. His scheme is an optimization of a full copying garbage collector that saves the cost of copying long-lived objects. His collector must still mark and sweep the entire memory space.

2.3.3 Write Barrier Performance

Hosking, Moss, and Stefanović at the University of Massachusetts [HMS92] evaluated the relative performance of various in-line write barrier implementations for a precise copying collector using five Smalltalk programs. They developed a language-independent garbage collector toolkit [HMD91] for copying, precise, generational garbage collection, which, like Ungar and Jackson, maintains a *large object space*. They compared the performance of several write barrier implementations: card marking using either in-line store checks or virtual memory, and explicit remembered sets, and presented a breakdown of scavenge time for each write barrier and program. Their research showed that maintaining the remembered sets explicitly outperformed other approaches in terms of CPU overhead for Smalltalk.

Zorn [Zor90a] showed an in-line write barrier exhibited lower-than-expected CPU overheads compared with using operating system page-protection traps to maintain a virtual-memory write barrier Specifically, he concluded that "...carefully designed in-line software tests appear to be the most effective way to implement the write barrier and result in overheads of 2–6%..."

In separate work [Zor90b] he showed properly designed mark-sweep collectors can significantly reduce the memory overhead for a small increase in CPU overhead in large LISP programs.

These results support the notion that using an in-line write barrier and non-copying collection can improve performance of garbage collection algorithms.

2.4 Contributions

Ungar and Jackson's collector provided a powerful tool for reducing the creation rate of tenured garbage by adjusting the promotion policy dynamically. I take this policy a step further and adjust the generation boundary directly instead. PARC's *Collector II* maintains such a threatening boundary, but they measured only the case where the time of the last collection was considered. I alter the threatening boundary dynamically before each scavenge which, unlike Ungar and Jackson's collector, allows objects to be *un-tenured*, and hence further reduce memory overhead due to tenured garbage. Unlike other generational garbage collection algorithms, I have adopted PARC's notation for immune and threatened sets, which simplifies specification of my collector over generational collectors.

In order to avoid compiler modifications, previous conservative collectors have used page-protection calls to the operating system for maintaining the write barrier. Recent work has shown program binaries may be modified without compiler support. Tools exist, such as *QPT*, *Pixie*, and *ATOM*, that alter the executable directly to do such tasks as trace generation and profiling [Dig91, LB94, SE94]. The same techniques may be applied to generational garbage collectors to add an in-line write barrier by inserting explicit instructions to check for pointer stores into the heap. Previous work has only evaluated in-line write barriers for languages other than C (e.g., LISP, Smalltalk, Cedar). I evaluate the costs of using an in-line write barrier for compiled C programs.

Generational copying collectors avoid destroying the locality of the program by compacting objects; conservative, non-copying collectors cannot

do this compaction. Even so, Zorn showed mark-sweep collectors can perform well, and malloc/free systems have been working in C and C++ for years with the same problem. However, in previous work I have examined the effectiveness of using the allocation site to predict short-lived objects [BZ93]. For the five C programs measured in that paper, typically over 90% of all objects were short lived and the allocation site often predicted over 80% of them. In addition, over 40% of all dynamic references were to predictable short-lived objects. By using the allocation site and object size to segregate short-lived objects into a small (64K-byte) arena, short-lived objects can be prevented from fragmenting memory occupied by long-lived ones. Because most references are to short-lived objects now contained in a small arena, the reference locality is significantly improved. In this document, I will discuss new work based upon lifetime prediction and store behavior to show future opportunities for applying the prediction model.

Albert Einstein observed that any theory should be as simple as possible to describe the observed facts, but no simpler. The same could be said of designs for complex software systems. The designer's task is to choose the simplest dynamic storage allocation system that meets the application's needs. Which system is chosen ultimately depends upon program behavior.

The designer chooses an algorithm, data structure, and implementation based upon the anticipated behavior and requirements of the application. Data of known size that lives for the entire duration of the program may be allocated statically. Stack allocation works well for the stack-like control flow for subroutine invocations. Program portions that allocate only fixed-sized objects lead naturally to the idea using explicit free lists to minimize memory fragmentation. The observation that the survival rate of objects is lower for the youngest ones motivated implementation of generational garbage collection. In all cases, observing behavior of the program resulted in innovative solutions.

All the work presented in this dissertation is based upon concrete measurements of program behavior. Program behavior is often the most important factor in deciding what algorithm or policy is most appropriate. While I present measurements in the context of the above three contributions, they are presented in enough detail to allow current and future researchers to gain useful insight from the behavior measurements themselves. Specifically, I present material about the store behavior of C programs, which has previously not appeared elsewhere.

Chapter 3

Implementation Costs

Any type of dynamic storage allocation system imposes both CPU and memory costs. The costs often strongly affect the performance of the system and pass directly to the purchaser of the hardware as well as to software project schedules. Thus, the selection of the appropriate storage management technique will often be determined primarily by its costs. This chapter will discuss the implementation model for garbage collection so that the experimental methods and results to follow may be evaluated properly. I will proceed from the simplest storage allocation strategies to the more complex strategies, adding refinements and describing their costs as I proceed. For each strategy, I will discuss the outline of the algorithm and data structures; then I will provide details of the CPU and memory costs. Initially, explicit storage allocation costs will be discussed and provide a context and motivation for the costs of the simplest garbage collection algorithms: mark-sweep and copy. Lastly, the more elaborate techniques of conservative and generational garbage collection are discussed.

3.1 Explicit Storage Allocation

Explicit dynamic storage allocation (DSA) provides two operations to the programmer: allocate and deallocate. Allocate creates uninitialized contiguous storage of the required size for a new *allocated object* and returns a *reference* to that storage. Deallocate takes a reference to an object and makes its storage available for future allocation by adding it to a free list data structure (objects in the free list are called *deallocated objects*). A size must be maintained for each allocated object so that deallocate can update the free list properly. Allocate gets new storage either from the free list or by calling an operating system function.

Allocate searches the free list first. If an appropriately sized memory segment is not available, allocate either breaks up an existing segment from the free list (if available) or requests a large segment from the operating system and adds it to the free list. Correspondingly, deallocate may coalesce segments with adjacent addresses into a single segment as it adds new entries to the free list (boundary tags may be added to each object to make this operation easier). The implementation is complicated slightly by alignment constraints of the CPU architecture since the storage must be appropriately aligned for access to the returned objects.

The costs of this strategy, in terms of CPU and memory overhead depend critically upon the implementation of the free list data structure and the policies used to modify it [DDZ94, GZH93, GZ93, KV85, MK88, ST85, Sta80, WW88, ZG94].

The CPU cost of allocation depends upon how long it takes to find a segment of the specified size in the free list (if present), possibly fragment it, remove it, and return the storage to the program. The CPU cost of deallocation depends upon the time to insert a segment of the specified address and size into the free list and coalesce adjacent segments. The total CPU overhead depends upon the allocation rate of the program as measured by the ratio of the total number of instructions required by the allocation and deallocation routines to the total number of instructions executed.

The memory overhead consists *entirely* of space consumed by objects in the free list waiting to be allocated (*external fragmentation* [Ran69])—assuming that internal fragmentation and the

space consumed by the size fields and boundary tags is negligible. *Internal fragmentation* [Ran69] is caused by objects that were allocated more storage than required (either to meet alignment constraints or to avoid creating too small a free-space element); careful tuning is often done to the allocator to minimize this internal fragmentation. The data structure required to maintain the free list may often be ignored because it can be stored in the free space itself.

The amount of storage consumed by items in the free list depends highly upon the program behavior and upon the policy used by allocate to select among multiple eligible candidates in the free list. For example, if the program interleaves creation of long-lived objects with many small short-lived ones and then later creates large objects, most of the items in the free list will be unused. Memory overheads (as measured by the ratio of size of the free space to the total memory required) of thirty to fifty percent are not unexpected [Knu73], which leaves much room for improvement [CL89]. The total memory overhead depends upon the size of the free space as compared to the total memory required by the program. This free list overhead is the proper one to use for comparing explicit dynamic storage allocation space overheads to those of garbage collection algorithms since garbage collection can be considered to be a form of deferred deallocation.

Often, both the CPU and memory costs of explicit deallocation are unacceptably high. Programmers often write specific allocation routines for objects of the same size and maintain a free list for those objects explicitly thereby avoiding both memory fragmentation and high CPU costs to maintain the free list. But, as the number of distinct object sizes increase, the space consumed by the multiple free lists become prohibitive. Also, the memory savings depend critically upon the programmer's ability to determine as soon as possible when storage is no longer required. When allocated objects may have have more than one reference to them (object sharing), high CPU costs can occur as code is invoked to maintain reference counts. Memory can become wasted by circular structures or by storage that is kept live longer than necessary to ensure program correctness.

3.2 Mark-Sweep Garbage Collection

Mark-sweep garbage collection relieves the programmer from the burden of invoking the deallocate operation—the collector performs the deallocation. In the simplest case, there is assumed to be a finite fixed upper bound on the amount of memory available to the allocate function. When the bound is exceeded, a *garbage collector* is invoked to search for and deallocate objects that will never be referenced again. The *mark phase* discovers reachable objects, and the *sweep phase* deallocates all unmarked objects. A set of mark bits is maintained, one mark bit for each allocated object. A queue is maintained to record reachable objects that have not yet been traced.

The algorithms proceeds as follows. First, the queue is empty, all the mark bits are cleared, and the search for reachable objects begins by adding to the queue all *roots*, that is, statically-allocated objects, objects on the stack, and objects pointed to by CPU registers. As each object is removed from the queue, its contents are *scanned* sequentially for pointers to allocated objects. As each pointer is discovered, the mark bit for the object being pointed to tested and set and, if unmarked, the object is queued. The mark phase terminates when the queue is empty. Next, during the sweep phase, the mark bit for each each allocated object is examined and, if clear, deallocate is called with that object. As a refinement, the implementor may use a set instead of a queue and may choose an order other than first-in-first-out for removing elements from the set.

Mark-sweep collection adds CPU costs (over explicit DSA) for clearing the mark bits and, for each reachable object, setting the mark bit, enqueuing, scanning, and dequeuing. In addition, the mark bit must be tested for each allocated object and each unreachable object must be located and deallocated. *Deferred sweeping* may be used to reduce the length of pauses caused when the collector interrupts the application. For deferred sweep, the collector resumes the program after the mark phase. Subsequent allocate requests test mark bits, deallocating unmarked objects until one of the required size is found. Deferred sweeping should be completed before the next collection is invoked since starting a collection when

memory is available is probably premature.

The first component of the memory cost for mark-sweep is the same as for explicit deallocation where the deallocation for each object is deferred until the next collection; this cost can be a very significant, often one and one half to three times the memory required by explicit deallocation [DDZ94, Knu73]. In addition to the size, a mark bit must be maintained for each allocated object. Memory for the queue to maintain the set of objects to be traced must be maintained by clever means to avoid becoming excessive [Boe94, Che70, SW67, Wal72]. A brute-force technique, to handle queue overflow, is to discard the queue and restart the mark phase *without* clearing the previously set mark bits [Boe95b]. If at least one mark bit is set before the queue is discarded, the algorithm will eventually terminate.

Virtual memory makes it attractive to collect more frequently than each time the entire virtual address space is exhausted. The frequency of collection affects both the CPU and memory overhead. As collections occur more frequently the memory overhead is reduced because unreachable objects are deallocated sooner but the CPU overhead rises as objects are traced multiple times before they are deallocated.

The two degenerate cases are interesting. Collecting at every allocation uses no more storage than explicit deallocation but at the maximal CPU cost; no collection at all has the minimum CPU overhead of explicit deallocation with a zero-cost deallocate operation, but consumes the most memory. The latter case may often be the best for short-lived programs that must be composed rapidly. The designer of the collector must tune the *collection interval* to match the resources available. Although this dissertation will not discuss it further, policies for setting the collection interval are an interesting topic in their own right, and there is much room for future research.

As mentioned earlier, during explicit dynamic storage deallocation, fragmentation can consume a significant portion of available memory, especially for systems that have high allocation and deallocation rates of objects of a wide variety of sizes and lifetimes. Other researchers have observed that the vast majority of objects (80–98%) have very short lifetimes—under one megabyte of allocation or a few million instructions [Wil95b]. This observation motivates two other forms of garbage collection: copying collection, which reduces fragmentation and sweep costs, and generational collection, which reduces trace times for each collection.

3.3 Copying Garbage Collection

Copying garbage collection marks objects by copying them to a separate empty address space (*tospace*) [FY69]. Mark bits are unnecessary because an address in tospace implicitly marks the object as reachable. After each object is copied, the address of the newly copied object is written into the old object's storage. The presence of this *forwarding pointer* indicates a previously marked object that need not be copied each subsequent time the object is visited. As each object is copied or a reference to a forwarding pointer is discovered, the collector overwrites the original object reference with the address of the new copy.

The sweep phase does not require examining mark bits or explicit calls to deallocate each unmarked object. Instead, the unused portion of tospace and the entire old address space (*fromspace*) becomes the new free list (*newspace*). Allocation from newspace becomes very inexpensive: incrementing an address, testing it for overflow, and returning the previous address. Collection occurs each time the test indicates overflow of the size of tospace. No explicit free list management is required.

Copying collection adds CPU overhead for the copying of the contents of each of the reachable objects. Memory overhead is added for maintaining a copy in tospace during the collection, but fragmentation is eliminated because copying makes the free list a contiguous newspace. Tospace may be kept small by ensuring that the survival rate is kept low by increasing the collection interval.

Copying collection can only be used where pointers can be reliably identified. If a value that appears to point to an object is changed to reflect the updated object's address, and that value is not a pointer, the program semantics would be altered.

3.4 Conservative Garbage Collection

Unlike with copying collection, *conservative collectors* may be used in languages where pointers are difficult to reliably identify. Conservative collectors are conservative in two ways: they assume that values are pointers for the purposes of determining whether an object is reachable, and that values are not pointers when considering an object for movement. They will not deallocate any object (or its descendents) referenced only by a value that appears to be a pointer, and they will not move an object once it has been allocated.

Conservative garbage collection requires a *pointer-finding heuristic* to determine which values will be considered potential pointers. More precise heuristics avoid unnecessary retained memory caused by misidentified pointers at the cost of additional memory and CPU overhead. The heuristic must maintain all allocated objects in a a data structure that is accessed each time a value is tested for pointer membership. The test takes a value that appears to be an address, and returns true if the value corresponds to the address pointing into a currently allocated object. This test will occur for each value contained in each traced root or heap object during the mark phase. The precise cost of the heuristic depends highly upon the architecture of the computer, operating system, language, compiler, run-time environment and the program itself. The Boehm collector usually requires 12 instructions on the DEC Alpha to map a 64-bit value to the corresponding allocated-object descriptor.

In addition to the trace cost, CPU overhead is incurred to insert an object into the pointer-finding data structure at each allocation, and to remove it at each deallocation. As with mark-sweep, deferred sweep may be used.

In addition to the memory for the mark bits previously mentioned for mark-sweep, conservative collectors require space for the pointer-finding data structure. On the DEC Alpha, the Boehm collector uses a two-level hash table to map 64-bit addresses to a page descriptor. All objects on a page are the same size. Six pointer-sized words per virtual-memory–sized page are required. The space for page descriptors is interleaved throughout dynamically allocated memory in pages that are never deallocated.

3.5 Generational Garbage Collection

Recall that generational garbage collectors attempt to reduce collection pauses by partitioning memory into one or more generations based upon the allocation time of an object; the youngest objects are collected more frequently than the oldest. Objects are assigned to generations, are promoted to older generation(s) as they age, and a write barrier is used to maintain the remembered set for each generation. The memory overhead consists of generation identifiers, tenured garbage, and the remembered set. Also, partition fragmentation can increase memory consumption for copying generational collectors when the memory space reserved for one generation cannot be used for the other generation. The CPU overhead consists of costs for promoting objects, the write barrier and updating the remembered set. Each of these costs are discussed in this section. An understanding of them is required to evaluate the results presented in the experimental chapters later in this dissertation.

The collector must keep track of which generation each object belongs to. For copying collectors, the generation is encoded by the object's address. For mark-sweep collectors, the generation must be maintained explicitly, usually by clustering objects into blocks of contiguous addresses and maintaining a word in the block encoding the generation to which all objects within the block belong [HMD91]. As objects age, they may be promoted to older generations either by copying or changing the value of the corresponding generation field.

Tenured garbage is memory overhead that occurs in generational collectors when objects in promoted generations are not collected until long after they become unreachable. In a sense, all garbage collectors generate tenured garbage from the time objects become unreachable until the next collection and memory leaks are the tenured garbage of explicit dynamic storage allocation systems. One of the central research contributions of this dissertation is to quantify the amount of tenured garbage for some applications, to show how it may be

reduced, and to show how that reduction can impact total memory requirements. Chapter 6 shows how tenured garbage from a generational collector mimics a memory leak bounded in time to the next scavenge of the generations containing that garbage.

In order to avoid tracing objects in generations older than the one currently being collected, a data structure, called the remembered set, is maintained for each generation. The remembered set contains the locations of all pointers into a generation from objects outside that generation. The remembered set is traced along with the root set when the scavenge begins. PARC's formal model called the remembered set a pointer augmentation and each element of the set was called a *rescuer*. This additional tracing guarantees that the collector will not erroneously collect objects in the younger (traced) generation reachable only through indirection through the older (untraced) generations. CPU overhead occurs during the trace phase in adding the appropriate remembered set to the roots, and in scanning each object pointed to from the remembered set.

A heuristic to reduce the size (and memory overhead) of the remembered set is often (indeed, universally) used: only pointers from generations older than the scavenged generation are recorded, but at the cost of requiring all younger generations to be traced. This heuristic makes a time-space tradeoff between increased CPU overhead for tracing younger generations to reduce the size of the remembered set based upon the assumption that *forward-in-time pointers* (pointers from older objects to younger ones) are rare. If objects containing pointers are rarely overwritten after being initialized, then the assumption would appear to be justified; however empirical evidence supporting this assumption is often not well supported in the literature when generational garbage collection is used in a specific language environment. Still, collecting all younger generations does have the advantage of reducing circular structures crossing generation boundaries [LH83].

The write barrier adds pointers to the remembered set as they are created by the application program. Each store that creates a pointer into a younger generation from an older one inserts that pointer into the remembered set. The write barrier may implemented either by an explicit in-line instruction sequence, or by virtual-memory page protection traps. The CPU cost of the instruction sequence consists of instructions inserted at each store. The sequence tests for creation of a forward-in-time inter-generational pointer and inserts the address of each pointer into the remembered set. The virtual-memory CPU cost consists of delays caused by page write-protect traps used to field the first store to each page in an older generation since the last collection of that generation. The cost of page-protection traps can be significant—on the order of 500-1000 microseconds [HM93], so there is motivation for investigating using an explicit instruction sequence for the write barrier. The in-line cost will be discussed in more detail in Section 5.2.

When three or more generations exist, updating the remembered sets requires the capability to delete entries.[1] The collector must ensure that unreachable objects discovered and deallocated from scavenged generations are removed from the remembered sets. A crude, but correct, approach is to delete all pointers from the remembered sets for the scavenged generations and then add them back as the trace phase proceeds.

Consider an n generation collector containing generations 0 (the youngest), to generation $n-1$ (the oldest). Before initiating the trace phase, suppose we decide to collect generations k and younger for some k such that $0 < k < n$. We delete from the rememberd set for each generation, t, such that $0 < t < k$, all pointers from generations s such that $t < s <= k$. As the trace proceeds, any pointer traced that crosses one or more generation boundaries from an older generation, s, to a younger generation, t, is then added to the remembered set for the target generation, t.

Another approach is to explicitly remove from each generation's remembered set all entries corresponding to pointers contained in each object as it is scanned. This deletion can occur during the mark phase, or as each object is deallocated during the (possibly deferred) sweep phase. The recent literature is not very precise about this cost [HMS92, HMD91], presumably because currently only generational collectors that use two generations are common. In this case, only one

[1] Extra entries in the remembered set will not cause the collector to fail, but it does increase trace costs and memory consumption

remembered set exists (for generation 0), and it is completely cleared only when a full collection occurs; precise remembered set update operations are not required.

Chapter 4

Experimental Methods

The results of this dissertation are empirical rather than analytical. The goal is to improve the performance of conservative generational garbage collection. In order to evaluate the effectiveness of this dissertation in meeting this goal, we must define what performance is, the sources of the performance data, how measurements are made, and which algorithms are being compared. This chapter will describe each of these subjects in turn and the chapters following will present results of experiments based on the material presented here.

4.1 Performance Metrics

Ultimately, research will lead to implementations, and those implementations will produce useful results that require computational resources when executed. If the results provide a benefit that outweighs the costs of the resources, then the implementations will be used and the research upon which they are based will have been justified. The more benefits outweigh the costs, the better. Thus, the metrics by which algorithms are compared should be highly relevant to their implementors and users. If they are not, then it is difficult to evaluate both the meaning and significance of the results of the research.

Two metrics result in direct costs to the user of a given implementation: CPU (time) costs, and memory costs. The previous chapter discussed, in detail, both of these costs for various techniques of garbage collection, and for generational and conservative garbage collectors in particular. These costs were chosen because they are the ones that have the most obvious and most direct relationship to real-life situations where time and money must be spent to obtain the resources necessary to use new algorithms. Higher performance algorithms result in lower costs to produce a given benefit.

CPU costs may be measured in a variety of ways. Ultimately, the user is most often concerned with the elapsed time to complete the tasks of interest. Unfortunately, this time may be affected by a variety of factors such as the processor architecture, clock speed, the number of processors used, bus bandwidth, memory hierarchy, and input-output device speeds. The degree to which each of these factors influences the task-completion time is highly influenced by the behavior of the tasks.

In order to simplify both the collection and presentation of results in this dissertation, only one CPU performance metric is considered: the total number of instructions executed. Usually, fewer instructions result lower execution time, and much of the compiler-optimization work uses the same metric. Time spent on cache misses and page-faults, while relevant as well, will be left for future work.

Memory costs will be measured in terms of total memory consumed, in bytes, by the application as it runs. Peak or maximum memory consumption is most relevant for single-tasking situations where virtual memory is not used. Other metrics for memory consumption such as the average, or median are applicable for virtual memory, or multi-tasking environments. As with the CPU metric, more complex measures of memory costs including paging behavior, locality of reference, and working-set size, while also of interest, will be left for future work. However, the cost of memory fragmentation will be included because it directly affects both maximum and average memory consumption but this cost will not broken out

Figure 4.1: Trace-driven Simulation.
An instrumentation program (ATOM or StCheck) reads an application program and instrumentation code (Instr Code or malloc) and produces an instrumented version of that program. The instrumented application (with its inputs) is executed to produce a trace of relevant events. This event trace is fed into a program (such as a GC Simulator) that simulates the algorithms being measured and produces appropriate statistics (GC or Store Statistics) from which CPU and memory costs are extracted.

The CPU and memory overheads of each of the programs were measured by using the technique

[2] Readers are invited to contact the author if they would like to contribute such programs for future research. Such contributions would be very welcome. Source code is not required if unstripped executables can be provided for the DEC Alpha running OSF-1.

Table 4.1: Sample programs measured.

Program	Description
CFRAC 6,000 lines 21 MBytes	Cfrac is a program that factors large integers using the continued fraction method. The input was a 28-digit number that was the product of two primes.
ESPRESSO 15,500 lines 182 MBytes	Espresso, version 2.3, is a logic circuit optimization program. The input was the largest example provided with the release code.
GAWK 8,500 lines 235 MBytes	GNU Awk, version 2.11, is a publicly available interpreter for the AWK report and extraction language. The input script formatted words in a dictionary.
GHOST 29,500 lines 101 MBytes	GhostScript, version 2.6.1, is a publicly available interpreter for the PostScript page-description language. The input was a large PhD thesis. GhostScript was not run interactively as is often done, but instead was executed with the NODISPLAY option, which causes results not to be displayed.
PERL 34,500 lines 33 MBytes	Perl 4.10, is a publicly available report extraction and printing language. The input script formatted the words in a dictionary into filled paragraphs.
SIS 172,000 lines 45 MBytes	SIS, Release 1.1, is a tool for synthesis of synchronous and asynchronous circuits. It includes capabilities such as state minimization and optimization. The input was one of the examples provided with the release (mcnc91/cse.blif), which attempted to reduce the depth of the circuit (`speed-up`).

of *trace-driven simulation*. A simulator for each of the algorithms being compared is executed using an event trace as input to produce performance statistics as output. An event trace is generated by executing a specially modified version of each application program. Modified applications are produced from the unmodified ones using an instrumentation tool. After performance statistics are generated, relevant statistics are extracted and processed to produce the desired CPU and memory costs. Figure 4.1 shows how trace-driven simulation was used to collect the results for Chapters 5 and 6.

Trace-driven simulation avoids the need to implement all the details of the algorithms being compared. The simulator implements an abstraction of the algorithm containing all the relevant details. A simulation model, incorporating a set of assumptions, determines what details are relevant. The trustworthiness of the simulation depends critically upon how faithfully the model matches a realistic implementation. Each of the simulations is deterministic, so only one execution of each application was required, and no statistical analysis (such as regressions, or error bars) were required—all measurements are exact. Each of the following three chapters containing results outlines the simulation model and the assumptions of that model to allow the reader to properly evaluate how strongly to trust their conclusions.

4.4 Algorithms Compared

In order to evaluate the relevance and significance of the results, each result must be compared against the existing state-of-the-art. Results should provide clear improvements over existing algorithms or should provide significant insights that assist future work. In each of the following chapters, a new algorithm, and the previous algorithms to which it is compared will be presented. All results will be presented in in comparison with those algorithms.

4.5 Summary and Outline of Later Chapters

The previous chapters discussed, in increasing detail, the general concepts necessary to understand and evaluate each of the principal contributions of this work. The next three chapters will present experimental results for each of these contributions.

Chapter 5 discusses the costs of an in-line write barrier for C programs. Chapter 6 introduces a new form of generational garbage collection and shows how it can substantially reduce the memory cost of tenured garbage without introducing excessive CPU overhead. Chapter 7 shows that the allocation site is an accurate predictor of short-lived objects and how store behavior may predict object lifetimes.

Chapter 5

In-line Write Barrier for C

Previous chapters have discussed, in general, the motivation, related work, contributions, implementation costs, and methods of this research. This chapter, along with the next two, mirrors this structure, but in more detail for each of the principal contributions of this dissertation.

This chapter concentrates on measurements of program behavior and the CPU costs of an in-line write barrier for generational garbage collection for C. The memory overhead of the write-barrier consists of the remembered set, which will be discussed in the next chapter. First, this chapter will discuss the motivation for investigating moving from a virtual-memory write barrier to an in-line barrier for C, and why program behavior is relevant. Next, I will discuss the specific costs of the write barrier and related work examining those costs. After that, I will present the methods used and the assumptions of the cost model. Next, I will present measurements showing the expected overhead of using an in-line write barrier based upon that model and the store behavior of several C programs. Last, implications for future research will be discussed.

5.1 Motivation

Generational collectors often use a write barrier to avoid tracing objects in older generations. Unfortunately, maintaining the write barrier can be expensive if not done carefully. One popular approach uses page-protection hardware to write protect pages that may contain objects in older generations. When a store occurs to such a page, a user-level trap handler is invoked in the garbage collector. The trap handler takes whatever actions are necessary to ensure that inter-generational forward-in-time pointers are recorded, unprotects the page, and resumes program execution.

Unfortunately, current operating systems are not designed to allow user programs to rapidly handle page-protection traps. Trap turnaround times of 744–915 microseconds have been measured by Hosking [HH93], which, on a 16 megahertz DECstation 3100, amounts to over 200 instructions per trap. Zorn [Zor90a] states trap costs of "over a millisecond" amounting to "...several thousand instructions." Thekkath and Levy [TL94] measured the trap times for several workstations and operating systems and observed times ranging from 69 to 2001 microseconds.

For comparison, I measured the trap times on a 266 MHz DEC AlphaStation 250 4/266 running the OSF-1 operating system version 3.2. The round-trip time for a user-level signal handler to field a page-protection trap was 38 microseconds[1], which represents sightly over 10 thousand instructions per trap on this machine. These measurements provide strong support for improving the costs of a write barrier.

This design makes sense for operating systems because page-protection traps either occur rarely (e.g., for access violations) or to retrieve non-resident virtual-memory pages into memory by incurring a disc transaction of at least 10 milliseconds. But, when used for garbage collection, page-protection traps will occur more frequently and will often not require a disc access because the page will already be resident. As technoligy advances, the situation gets worse because processor speeds are improving more rapidly than disc speeds.

[1] Excluding all ten instructions in the user trap-handler

Although there has been some recent work to improve the speed of operating-system traps [TL94], another solution is to provide user-level access to virtual-memory dirty bits [Sha88, §6.6.8, pp 129-133]. This solution avoids the need for the trap handler in the garbage collector to be invoked by the operating system. Shaw added three operating system calls: one for clearing the virtual-memory dirty bits for each page, another for reading them, and another that combines both operations. The collector clears the dirty bits after pages have been scanned, and re-examines them during each subsequent scavenge. The operating system must be modified to cooperate with the collector for maintaining the dirty bits and the virtual memory. Unfortunately, dirty-bit operations are not yet widely supplied by the operating-system interface.

How well a virtual-memory write barrier performs depends highly upon program behavior. Only stores that create inter-generational forward-in-time pointers must be trapped. Languages that have few such stores will not incur high overheads even though traps are expensive. Forward-in-time pointers can only be created when an existing object is over-written. Programs in highly applicative functional languages, such as ML, would create few such pointers, and would perform well, regardless of the write barrier implementation. Some Smalltalk programs have low overheads as well [HH93].

The costs of a write barrier for garbage collection of C and C++ programs may be much worse. As mentioned earlier, garbage-collection for such programs is desirable as an option and several researchers have made significant effort to provide that option [Bar90, Boe94, Det93]. Providing an efficient cost-effective virtual-memory write barrier is difficult; the only currently (July 1995) portable widely available implementation known to the author is [Boe94].

Bartlett avoids the need for a write barrier by having the user program provide routines for tracing objects in the older generation. Boehm's implementation provides hooks for generational collection, but it is not enabled by default (See MPROTECT_VDB in [Boe95a]). In addition, Boehm [Boe95a] has noted that VM page traps may not detect pointer stores performed by the operating system during system calls.[2]

[2] Presumably pointers stored by such calls are rare, but Boehm implemented a wrapper for the **read** system call to illustrate how to deal with this problem.

Research on aspects of program behavior for C or C++ that pertain to the write barrier has never been done before and is important for people wishing to implement generational collection for C or C++.

5.2 Write-barrier Costs

The CPU overhead of an in-line write barrier depends upon precise program behavior. Recall from Section 3.5 (page 17) that the write barrier adds elements into the remembered set. For an in-line write barrier, a sequence of instructions (the store-check) must be added before or after each store instruction of the program. Recall also that only inter-generational forward-in-time pointers must be added. Thus, only a small fraction of total stores will actually result in additions to the remembered set. The size of this fraction and its cost depends upon several aspects of program behavior.

For the most general case, each store check must:

1) verify the store is a non-initializing store (optional optimization),
2) ensure the value being stored is a pointer,
3) ensure the pointer corresponds to a dynamically allocated object,
4) determine the objects (if any) corresponding to the location (source), and value (target) of the pointer being stored,
5) ensure the generation of the source is older than the target, and
6) add the address being stored into the remembered set.

Also recall that at the beginning of each trace phase, the garbage collector must enqueue the elements of the appropriate remembered set, and then trace those objects (pages 14,17).

All of the above operations above may be optimized in various ways, depending upon the architecture of the CPU, compiler, run-time environment, operating system and programming language. For an extreme example, consider the case

where each object is tagged with its type and copying generational garbage collection is used. The non-initializing pointer test (operations 1,2,3) may often be eliminated by static information available at the time the store instruction is compiled, and if not eliminated, only require a single instruction test the type tag. Operations 4 and 5 consist of a compare of the source and target addresses, and operation 6 is an insertion into a hash table. There are numerous other possibilities depending upon how restrictive the architectures are and how precise the remembered sets are.

Hölzle [Hö93] stated that just two instructions are required for the SELF language where *card marking* [WM89a] is used for the remembered set. Card marking trades reduced store check costs for increased tracing costs. Remembered set elements consist of cards (typically 128 bytes each) to scan for pointers to objects rather than than single pointer locations. Hölzle uses a shift instruction to calculate the offset into a byte map from the address of the store, and a store to clear that byte. A dedicated register points to the base of the byte map that corresponds to a single remembered set for a two-generation collector. This cost presumes that a single byte map exists for the entire heap. For heaps that are too large to allocate a static byte map in advance, setting and checking multiple byte maps would add additional overhead. Their method defers all but operation 6 until the trace phase.

How the work of the write barrier is partitioned between the store check and the trace phase will affect performance. The cost during tracing depends upon the number of modified locations rather than the number of stores and its affect upon the cache and page locality of the program is probably different. For card-marking schemes, the spatial pointer density can increase costs as well; a program that stores to one pointer on each card in the entire heap would cause a generational collector to scan every object in the heap. Hosking and Moss [HM93] designed a hybrid scheme that transfers pointer locations from marked cards to the remembered set during the next trace phase. When results are presented for write-barrier costs, the reader should be careful to ensure that this partitioning is clear and that all costs have been presented.

Conservative collection can interact with the write barrier to increase its cost. For operations 4 and 5 each store instruction must determine the object corresponding to the source and invoke the pointer-finding heuristic (page 16) for the target. Interior pointers (see page 8) make the pointer-finding heuristic slightly more complex because it must locate the bases of objects. Derived pointers (p. 2.2.3) can prevent recognition of pointers to reachable objects entirely.

5.3 Methods

As mentioned in Chapter 4, several C programs were instrumented to produce traces of program behavior. The traces, consisting of sequences of store and allocation events generated as the program ran, were analyzed and that analysis was used with a performance model to produce the expected CPU overheads for an in-line write barrier.

Program instrumentation used to be a much more difficult problem than it is today. When this research began, Larus provided a tool called AE, for *abstract execution* on the Sun SPARC. AE used the object files for a program with a set of interesting events to produce a C program, an abstraction of the original, that generated the same sequence of events. That C program was then linked with instrumentation code to produce the desired data.

The events of interest were store events and memory allocation and deallocation events. For each store event, I needed to know the instruction performing the store, the location of that instruction, the value being stored (register contents), its size (word or quadword), location (effective address), and the value being overwritten (memory contents). Unfortunately, with AE, the C program generating the events was an abstraction and it did not provide a way to obtain the contents of a memory location being overwritten.[3]

As an alternative to AE, I wrote an instrumentation program (StCheck) that replaced all store instructions in an object file with calls to an instrumentation routine followed by the original store (see the right-hand side of Figure 4.1 on page 20). The new object file was then linked with a special version of the memory allocation routines (e.g., malloc and free) and the store instrumentation routine. Each application program

[3] Knowing when a pointer is overwritten allows the simulator to verify object reachability.

was instrumented and executed to produce a set of store statistics.

An instruction profiling tool, Pixie [Dig91], was used to collect the the total number of instructions executed by each instrumented program.

Store statistics and the pixie output were then fed into PERL-language scripts that incorporated a write-barrier model and subtracted the instrumentation overhead to extract the expected CPU overhead.

5.4 Results

This section discusses two results regarding an in-line write barrier. First, I show how the store behavior of real-world C programs and a typical computer architecture influence the write barrier. Second, I show how that behavior influences total CPU overhead for a 20-instruction write barrier.

5.4.1 Store Behavior

Recall from Section 5.2 (page 24), that the CPU architecture plays a role in the costs of the write barrier. All programs were executed on a DEC Alpha. The Alpha is a 64-bit RISC machine, where pointers occupy eight bytes each. The only instructions that operate upon memory are 4 or 8-byte loads and stores. Memory accesses not aligned by the size of their operand incur a trap. Special instructions exist for doing unaligned, byte, or word accesses. This architecture is fairly typical of modern RISC machines except that it uses 64-bit rather than 32-bit pointers.

As mentioned in Section 5.2 several tests must be performed by the write barrier at each store instruction in the program. In order to compute the costs associated with each of the operations required, the program store behavior of each of the instrumented programs was collected as shown in Table 5.1.

First, observe that the number of store instructions as a percentage of total instructions executed ranged from 5.24% to 9.26%—one in every 11 to 19 instructions. Even if only two instructions must be added at every store, an in-line write barrier would add 10 to 18 percent to the execution time of the program; thus some form of static analysis is needed to reduce this cost. Fortunately, there are several easy heuristics.

Table 5.1: Program Store Behavior.

Program	Exec. Instr. (10^6)	Store Instr. (%)	Filter Instr. (%)	Forward Instr. (%)
CFRAC	1471	6.55	0.16	0.000
ESPRESSO	2362	5.24	0.60	0.154
GAWK	1731	9.26	2.52	0.152
GHOST	1646	7.39	1.81	0.127
SIS	487	6.35	1.26	0.160

The CPU overhead for the write barrier for several programs above depends upon their store behavior. All percentages are of total instructions executed (Exec. Instr.). The Filter Instr. column shows the percentage of total instructions requiring a filter to check for forward-in-time pointers. This percentage is small compared to the store instruction percentage (Store Instr.), which is itself a small percentage of total instructions. Forward Instr. shows the percentage of total instructions that required a forward-in-time pointer to be inserted into the remembered set.

The store instruction itself provides useful information for static analysis. For example, only heap stores are interesting. The C compiler allocates local variables on a *stack* and accesses them using a dedicated register called the stack pointer. If only stack (and not heap) objects are accessed by the stack pointer, then any store instructions using a displacement from the stack pointer may safely be ignored by the write barrier.

Note that pointer indirection is a frequent operation, and that accesses are often significantly (often 100%) slower for unaligned than aligned memory operations. On the Alpha, the overhead is even worse because a trap will occur. Architectures often have separate instructions for aligned and unaligned stores. It seems valid to assume that the compiler will avoid creating unaligned pointers, and hence avoid generating unaligned pointer store instructions. Thus, any unaligned store instructions can also be ignored statically.

Note also, that pointers have a fixed size. Many computer architectures also encode the size of the entity being stored into the instruction. Any store of a value of a different size than a pointer may also be ignored safely.

The Filter Instr. column of the table shows the percentage of total instructions executed that were not able to be eliminated by one of the three simple static analysis heuristics described above. As the table shows, this value is a small percent-

Figure 5.1: CPU Overhead of an In-line Write Barrier.
An in-line write barrier increases the total instructions executed by the sample programs up to an additional 50%.

The CPU overheads ranged from from 3% for CFRAC to 50% for GAWK. Only one instruction in 625 needed to be checked for CFRAC as compared to one in 40 for GAWK. GAWK also had both the highest percentage of store instructions and the largest heap size of any of the programs measured. There was also a high correlation between ranking of the programs in terms of CPU overhead and maximum heap size (see Table 6.1, page 39) as well as the ratio of store instructions to total instructions.

The CPU overhead for the write barrier using only the simple static heuristics mentioned

earlier and a 20-instruction write barrier is likely to be too high for most C or C++ applications. This result contrasts sharply with Smalltalk, where store checks caused an overhead of less than 2%.[5] [HM93] For LISP, Moon [Moo84, p. 243] estimated from 10–150% overhead for a software store-check.[6]

5.5 Implications

I restricted the heuristics available and chose a high write-barrier cost based upon the assumption that no compiler support would be available from C++ compilers, and that executable-file instrumentation tools would be of limited capability. If these assumptions are relaxed slightly, improvements appear likely on two fronts: increasing the effectiveness of static analysis in reducing the number of annotated stores, and reducing the number of instructions for the barrier.

These measurements did not apply a heuristic for removing any initializing stores. In the case of C++, statically recognizing initializing stores for heap objects may be quite possible because objects are often allocated by a class constructor, which also initializes its object. Techniques such as data-flow analysis used by compiler optimizers may be very effective at detecting such stores. Even if the compiler cannot be modified for this purpose, recent work at DEC WRL indicates that the capabilities of executable file translators have advanced very significantly since work for this dissertation began [SW94]. The author has done some preliminary measurements showing that initializing stores could be a significant fraction (greater than 30%) of all heap stores for the C programs above. Precise measurements of rates of initializing stores for C++ programs would be very useful future work.

In this work, no static type information was used to reduce the number of annotated stores. The language designs for some languages, such as Modula-3, significantly increase the static type-checking capability of the compiler. Any assignment that could be recognized as a non-pointer store at compile-time need not be annotated. Determining the effectiveness of such type tests upon improving the "aligned pointer-sized heap store" heuristic above as well as reducing the instruction count for the barrier could be fruitful avenues of future work.

Reducing the number of instructions for the write barrier is essentially a form of code optimization. Code optimizations has been researched very heavily in the last few years [BGS94]. Placing hash table and object descriptor addresses into dedicated registers may provide a significant reduction in the code-sequence required. My estimate on the number of instructions assumed all pointers could be interior pointers. A simpler, faster pointer-finding heuristic may be possible for stores of pointers that point only to the lowest address of an object.

I have assumed that the entire write barrier was implemented in-line rather than by using some of the tradeoffs made by card-marking schemes that move some of the work into the trace phase of the collector. This result shows that making such a tradeoff even for an in-line write barrier may be appropriate—by using Moss's [HMS92] *Sequential Store Buffer* for example.

This section investigated one technique for improving the performance of generational garbage collection. I have talked about how the mechanism of an in-line write barrier affects CPU overhead in the context of C. In the next chapter, I will investigate a new method for improving the memory performance as well.

[5] For the *remsets* implementation, Hosking and Moss reported 654,245 checked stores in 42 seconds for the *Interactive* benchmark on a 16.67 MHz DECstation 3100. I assume 20 cycles per check.

[6] Moon stated 17–25 microseconds between stores and 2.5–25 microseconds for each store check.

Chapter 6

Dynamic Threatening Boundary

This chapter discusses an improvement to generational garbage collection that reduces its memory consumption by more effectively reclaiming tenured garbage. The first section discusses the advantages of using policy to define mechanism over the reverse approach. In the next section, I motivate and discuss an overview of my algorithm in the context of other related work. Sections 6.4.1 and 6.4.2 discuss, in detail, the mechanism and its implementation. Sections 6.4.3 and 6.4.4 derive one policy using this mechanism. Lastly, Sections 6.5 and 6.6 present the experimental methods and results.

6.1 Policy versus Mechanism

The purpose of a garbage collector is to recover the maximum amount of memory with the least expenditure of CPU time. Recall from Section 2.2.2 (p. 7) that generational garbage collectors exploit the property that younger objects have a higher probability of becoming garbage than older ones. The collector can increase its productivity (instructions executed per byte reclaimed) by preferentially scavenging younger objects over older ones since fewer objects will be traced and a higher percentage will be garbage. Generational collectors work by using a mechanism to implement a policy for dividing objects into either young or old generations based upon their age.

Unfortunately, policy is often dictated by implementation constraints upon the mechanism rather than desired performance goals. An easy-to-implement mechanism that performs acceptably will often be used no matter what policy it follows. The implementation defines the policy rather than the other way around. Ideally a policy would be selected to meet a performance objective and that policy would suggest a set of possible mechanisms, one of which would be chosen for implementation.

Using the implementation to define the policy works well until the implicit assumptions change, causing the performance to degrade to an unacceptable level. Often, new implementation techniques are tried that define new policies until the performance is again acceptable. The problem with this approach is that the underlying interaction between program behavior and policies never becomes clearly understood. Neither behavior nor policies need be examined to implement a bottom-up mechanism-to-performance trial-and-error technique.

Wilson [Wil95a] has conducted a very recent survey of the literature that shows how this approach influences explicit dynamic storage allocation design. For example, suppose the mechanism of a doubly linked list is used to maintain a free list of available blocks of storage, and a pointer is used to point to the next block in the list. Whenever an item is freed, it is inserted into the list before the block being pointed to. Whenever an item is allocated, the pointer is advanced until a block large enough to meet the request is located. Then the block is split yielding a block of the correct size, and a fragment. The block is returned to the program, and the fragment remains in the free list.

The allocation routine must decide how meet the request: choose among the blocks in the free list large enough to meet the request, whether to split the block and how, and when to get more memory from the operating system. Which choice it makes defines an allocation policy. The policy defined by the mechanism depends critically

upon minute design decisions made by the implementor. For example, how the pointer is manipulated and the ordering of the list can define policies of first-fit/lowest-address, first-in/first-out (FIFO), last-in/first-out (LIFO), or next-fit. Each of these policies has an impact on CPU and memory performance. Trivial changes to the mechanism can have dramatic affects upon the policy, and by implication, performance. For example, Wilson states that using first-fit/lowest address yielded 22% fragmentation while LIFO/next-fit yielded 59%.

Generational garbage collectors must make similar implementation, mechanism, and policy choices for when to collect, how to implement the write barrier, and selecting which objects to collect. One of my principal goals for this dissertation was to find more flexible ways to improve the performance of garbage collection. Instead of letting mechanism define policy, I chose to work in the opposite direction: define what performance is important, determine what behaviors affect that performance, use that behavior to define a policy to improve it, design a mechanism to implement that policy, and finally, propose an implementation for that mechanism.

With this general goal in mind, I began examining the performance issues important for generational garbage collectors: CPU and memory overhead. Then, I realized that generational collectors must have a policy for deciding what to trace at each collection: selecting the immune set. I sought a mechanism that provides for the most flexible policy choices: the dynamic threatening boundary. Then I examined various policies, both existing and new, to see which ones were the most promising. Finally, I conducted experiments to see how program behavior affected the performance of each of these policies.

6.2 Motivation

All generational collectors trade space for time. By limiting the number of objects traced, the CPU time lost to tracing is improved, but at the cost of increased memory consumption due to tenured garbage. Specifically, all generational algorithms occasionally promote objects that eventually become tenured garbage. Tenured garbage eventually needs to be reclaimed in some way, otherwise it will cause excessive memory consumption and an associated degradation in performance due to decreased reference locality.

The simplest solution to this problem is to occasionally collect all the older objects at a time when a long pause, possibly accompanied by many page faults, will not be too disruptive. Another solution is to set the garbage collector's promotion threshold (i.e., the age at which objects are promoted) to a fixed value that minimizes the tenured garbage. Unfortunately, both identifying acceptable times for long pauses, and identifying an appropriate promotion threshold, varies from one program run to the next and even during the lifetime of a single program. As a result, Ungar and Jackson [UJ92] propose a method called *feedback mediation* (FM), which provides a partial solution to this problem by adjusting the promotion threshold for old objects based upon a pause-time constraint and object lifetime demographics. Their collector reduces tenured garbage by reducing the rate of object promotion. However, once objects are promoted using their method, the problem of reclaiming them still remains.

6.3 Overview and Scope

In Section 6.4.1, I describe a mechanism that extends existing generational collection algorithms by allowing them to reclaim tenured garbage more effectively. In particular, my *Dynamic Threatening Boundary* (DTB) mechanism divides memory into two spaces, one for short-lived, and another for long-lived objects. Unlike previous work, my collection mechanism can dynamically adjust the boundary between these two spaces either forward or backward in time, essentially allowing data to become untenured.

To implement the DTB mechanism, all pointers that point forward-in-time must be recorded in the remembered set, unlike standard generational collectors, where only forward-in-time pointers that cross the generation boundaries (*inter-generational pointers*) are recorded. In Section 6.6.1, I measure the overhead of maintaining the DTB remembered set and the cost of processing it during collection. My measurements of allocation-intensive C programs indicate that the space overhead of the DTB ranges from 4–15% of the maximum storage required by the program,

while the CPU overhead of maintaining the DTB ranges from 0–7% of total execution time (as measured by the number of instructions executed).

The techniques proposed should be most successful in two classes of languages: languages where pointers tend to account for a small fraction of the total memory allocated, and languages where most pointers do not point forward-in-time (i.e., where programs perform few destructive update operations, such as Standard ML). My measurements in Section 6.6 indicate that C may belong in the first category.

Once the dynamic threatening boundary mechanism is available, it provides significant flexibility to the collector implementation and clearly separates issues of policy from implementation. In Section 6.4.4, I explore one policy, using the DTB mechanism, that attempts to reduce a program's tenured garbage. My policy extends Ungar and Jackson's feedback mediation policy by taking advantage of the flexibility provided by the DTB. My results show that this policy is more successful than feedback mediation at reducing tenured garbage and results in smaller program heaps when such tenured garbage exists, even when the additional space overhead of maintaining the DTB is taken into account.

In Section 2.3 (p. 9) I discussed the work my idea is most closely based upon. My work deals with generational garbage collection, which is a subset of more general selection policies. The general class collects only a subset of all allocated memory by using a selection policy to decide what to collect. Generational collectors use an age-based selection policy. Other selection policies are possible.

Wilson and Moher's *Opportunistic Collector* [WM89b] allocates objects created since the last collection in chronological order in memory. By selecting an appropriate address, only objects allocated since a specific time may be selected for promotion. However, once their collector has reclaimed objects from this new-object area, a different promotion policy must be followed because surviving objects are no longer in chronological order. My algorithm preserves the object's allocation time for all objects, not just new ones, so mine may select among surviving objects of any age as well.

When age is not a reliable indicator of garbage non-generational methods must be used. Hudson and Moss [HM92] describe a *mature object space* that is collected incrementally based upon object connectivity rather than age. Likewise, Hayes [Hay91] shows that when certain *key objects* die, they may indicate other unused ones as well. Like generational collectors, mine could eliminate objects from age-based collection by promoting them to mature or key object space, where they would be collected by other algorithms once they age enough.

The DTB mechanism I describe is an extreme case of a collection implementation that allows multiple generations (e.g., [CWB86, HMD91]). As the number of generations grows to the number of live objects, the two concepts merge. From this perspective, my proposed collection policy (DTB_{dg}) represents a policy to select which generation to collect. Also, this previous work has not quantified the overhead of maintaining large numbers of generations, either in terms of CPU cost or memory overhead as I shall. Note that policies for deciding what to collect are completely independent of deciding when to collect. Because both affect performance, the distinction is often confusing: this research investigates a very general age-based mechanism for partitioning what to collect and one policy using that mechanism.

6.4 Dynamic Threatening Boundary Collector

In this section I describe the DTB mechanism and a policy for a collector that uses it to reduce tenured garbage. I first describe how it provides the capability to dynamically adjust the boundary between old and new objects based upon the object allocation time. Next, I describe the new implementation issues raised by my mechanism. Then I discuss how the threatening boundary selection policy influences tenured garbage and pause times. Finally, I describe one policy that makes use of the DTB mechanism to trade available pause time for reduced tenured garbage.

6.4.1 The Dynamic Threatening Boundary Mechanism

Recall from Section 2.3.1 (p. 9) that we can view a generational collector as partitioning the allocated

space into threatened and immune sets. Threatened objects are those that the collector traces to find unreachable objects and reclaim them. Immune objects are ones that will not be traced in a given collection. The selection criteria for these sets distinguishes various collection algorithms.

Consider how a traditional generational collector selects its threatened and immune sets. The threatened set contains those objects that have survived fewer than a specified number of collections—typically one or two [App92, Fra92, WM89b]. The root objects and all objects in older generations are immune. The *threatening boundary* divides the young threatened objects from the old immune objects. Each time the garbage collector is invoked, its policy sets the threatening boundary to the time of the kth previous collection, where k is a small integer constant. Scavenging the kth older generation corresponds to temporarily choosing a threatening boundary to the age corresponding to the kth previous generation boundary. Generation boundaries simply constrain the set of allowable threatening boundaries.

My mechanism eliminates generation boundaries. Instead, an explicit threatening boundary is established at the beginning of each collection. This boundary allows the collector to be much more flexible in choosing policies for selecting the threatened set.

Figure 6.1 illustrates how the dynamic threatening boundary collector compares with other generational collectors. This figure shows a memory space divided into two generations. Age proceeds from youngest objects at the top of the page to the oldest at the bottom, whereas birth time increases in the other direction. Objects (in rectangles) are labeled in sequence by age. Arrows (labeled in lower case), indicate pointers between objects; heavy arrows indicate forward-in-time pointers.

For a generational collector, only pointers f and h must be recorded by the remembered set for Generation 0 because otherwise object H would be incorrectly deallocated by a scavenge of Generation 0. While the garbage objects B and E would be scavenged, objects J, K, and F would not; they are tenured garbage. Object F illustrates the phenomenon of *nepotism*: it remains alive even though it is threatened and unreachable because the tenured garbage points to it. Once promoted, tenured

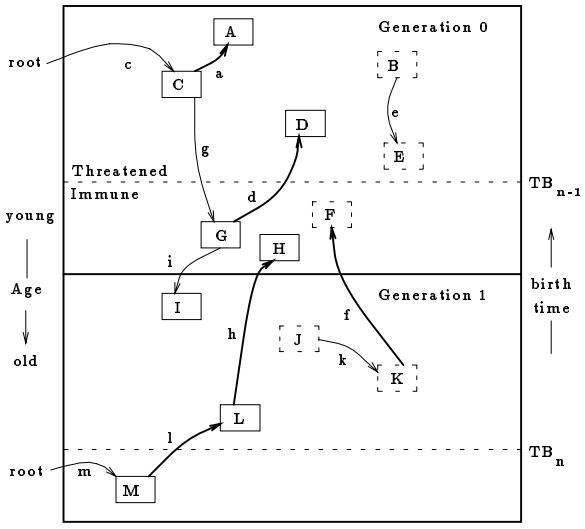

Figure 6.1: Dynamic Threatening Boundary vs Generations.
The generational collector above divides memory into two generations, one young and one old. A dynamic threatening boundary collector adjusts a threatening boundary that may move between scavenges, say from TB_{n-1} to TB_n. Objects are shown ordered by birth time for exposition only; the actual implementation may maintain object locations in any order. Objects with solid outline indicate reachable live objects and those with a dashed outline indicate garbage.

garbage requires a complete scavenge of its generation to be reclaimed, in this case, Generation 1. A non-generational collector always collects all generations and so would collect all the garbage objects (B, E, F, J, and K) at the cost of tracing the entire memory space.

For a dynamic threatening boundary collector, a threatening boundary (shown by a dashed line at TB_{n-1}), divides the memory into *threatened* and *immune* spaces. Because the threatening boundary can be changed at the beginning of each scavenge, all forward-in-time pointers must be maintained in a single remembered set (pointers a, d, h, f, and l). At scavenge time only pointers that cross the threatening boundary are traced (pointers d). On a later scavenge, the collector is free to set the new threatening boundary to any desired time, say at TB_n. Unlike the generational collector, objects J, K and F become *untenured*, and will be reclaimed. Object L remains alive because pointer l references it from the remembered set.

6.4.2 Dynamic Threatening Boundary Implementation

In this section, I describe the implementation of the DTB mechanism. First, I state my assumptions; next I describe the implementation at a high level; and finally I discuss aspects of the implementation that are different from existing generational mechanisms.

For the purpose of this discussion, I assume that I am implementing the DTB mechanism in the context of a non-copying algorithm. This assumption implies that objects are not relocated when collected and that the birth time of an object cannot be encoded in its address (as is the case in most copying algorithms). Thus, I assume that every object has a birth time field associated with it that is set when the object is allocated and never modified. I also assume that a data structure exists (the remembered set, which I describe in detail below), that indicates the location of every forward-in-time pointer. That is, every pointer stored in an object, o_1, that points to an object, o_2, such that the birth time of o_1 is less than the birth time of o_2 (o_2 is younger than o_1).

To understand how the DTB mechanism works, suppose that I am about to do a garbage collection and some policy has determined some specific threatening boundary. First, I augment the normal root set (i.e., registers and stack) with some of the elements in the remembered set. Specifically, I scan the remembered set and find all elements of it such that the forward-in-time pointer from the set points from an object born before the threatening boundary to an object born after the threatening boundary. The augmented root set now contains pointers to all objects in the threatened set that are reachable from the root set or immune set. I transitively traverse these roots marking all reachable objects in the threatened set. To complete the collection, I must then sweep the entire memory searching for unmarked objects that have a birth time later than the current threatening boundary. In practice, such a sweep can be deferred [Zor90b], reducing its performance impact.

The implementation of a dynamic threatening boundary mechanism relies mostly upon technology already available for other generational collectors. Here I describe new implementation issues raised by my mechanism and how they affect performance. These issues are maintaining object birth times, the effect of the remembered set on memory consumption, the write barrier, and tracing.

Object birth times must be available to determine the threatened set and to allow the write barrier to maintain the remembered set. The most straightforward implementation maintains a word per object. In environments where many small objects are allocated, objects may be co-located into pages (or areas) sharing the same birth time. Birth time may be chosen to be any appropriate metric of the granularity desired. One metric corresponding closely to existing generational collectors is the number of collections preceding the object allocation. A finer grained metric might be the total number of bytes allocated before the object was allocated.

Typically, a remembered set is maintained for each generation except the oldest; since my mechanism has only two generations, and the boundary between them moves, it uses a single remembered set instead. Generational collectors record only forward-in-time pointers that cross generation boundaries whereas mine records all forward-in-time pointers. My remembered set is larger; I show how much larger in Section 6.6.1.

Recall from Section 5.2 (p. 24) that one component of the CPU costs for the write barrier is the time spent adding a pointer to the remembered set. The barrier cost for the DTB mechanism is the same, except the birth times of the objects are compared instead of the generations. My algorithm performs this remembered set insertion more often than other generational collectors since I insert all forward-in-time pointers rather than just those that cross the threatening boundary. I explain the cost in detail in in Section 6.6.

Like all generational collectors, each element of the remembered set must be examined as an additional root during the trace phase of each scavenge. My remembered set is larger, and I must perform an additional test to ensure I only trace objects that are born after the current threatening boundary. In Section 6.6 I show how much my mechanism would increase CPU compared to a traditional generational collector.

6.4.3 How Policies Affect Collector Performance

The choice of the threatening boundary affects both the CPU time spent scavenging and the memory wasted by tenured garbage. For a given collection interval, a young threatening boundary results in short trace times at the expense of more tenured garbage. An older threatening boundary wastes more CPU time tracing more of the live objects, but saves memory because older unreachable objects are reclaimed sooner.

Figure 6.2 shows how these values are related. The vertical axis is storage consumed (in bytes) and the horizontal axis is execution time (CPU instructions executed). Consider how a full garbage collection behaves. Periodically, at time t_i, a scavenge is triggered. The collector traces all the live storage and reclaims the rest. For example, at time t_1, Mem_1 bytes of storage were in use before the scavenge; the collector traced $Trace_1$ bytes, which included all the live bytes L_1. All the remaining bytes were reclaimed as shown by the curve dropping vertically to L_1

A generational collector scavenging at time t_{n-1} would only trace objects born after a fixed-age threatening boundary TB_{n-1}. This results in shorter pause times due to less storage traced, $Trace_{n-1}$, at the cost of more storage surviving, S_{n-1}. The difference between S_{n-1} and L_{n-1} is the tenured garbage.

At time t_n, the dynamic threatening boundary mechanism must select a threatening boundary TB_n before initiating scavenge n. The farther back in time TB_n is, the more storage will be traced, and the more garbage reclaimed.

6.4.4 A Policy to Reduce Tenured Garbage: DTB$_{dg}$

One important policy for all generational collectors is when to *promote* objects from threatened space to immune space. A non-generational collector always collects all data, which corresponds to selecting the threatening boundary to be zero at every collection. I call such a collector FULL, and note that FULL never tenures any garbage.

A *fixed-age* generational collector promotes all objects that survive k collections, where k is a fixed value. This choice corresponds to setting the threatening boundary a fixed distance backward in time. If $k = 1$, (i.e., the FIXED1 policy used in my results) then this policy corresponds to tenuring objects as soon as they survive a single collection.

Feedback mediation (or FM) is more sophisticated than a fixed-age policy. Instead of blindly setting the threatening boundary a fixed distance in the past, feedback mediation only advances the threatening boundary when a pause-time constraint (as defined by the number of bytes that are traced) is exceeded. In particular, if the current scavenge traces more bytes than a specified maximum, $Trace_{max}$, the threatening boundary is advanced, otherwise it is not and no objects are promoted.

To advance the threatening boundary, the FM algorithm maintains a table of object demographics as it scavenges. The table classifies the currently scavenged objects by birth time into categories (or birth-time regions) and identifies how many bytes of objects are alive in each birth-time region. When the threatening boundary must be advanced, this table is scanned backwards starting at the current time. The scan accumulates the number of live bytes in each region until $Trace_{max}$ bytes is reached. This point determines where the new threatening boundary is set. In feedback mediation, demographics data are not preserved from one scavenge to the next, and information about objects born before the current threatening boundary are neither available nor used.

I seek a policy that reduces tenured garbage more effectively than feedback mediation. My policy, DTB$_{dg}$ (DTB demographic) attempts to do this by using the ability of the mechanism to dynamically adjust the threatening boundary to any desired value. The policy attempts to meet two goals: control pause times and minimize tenured garbage. Like feedback mediation, the policy attempts to meet the first goal by explicitly adjusting the threatening boundary forward to reduce pause times when it traces more than $Trace_{max}$ bytes. However, to meet the second goal more effectively than feedback mediation it moves the boundary backward in time when it traces fewer than $Trace_{max}$ bytes. To do this adjustment, it preserves demographic information from one collection to the next and uses it in the same manner as feedback mediation, but without the constraint against moving the boundary backward.

My policy has a couple of additional refinements. First, it attempts to predict how much

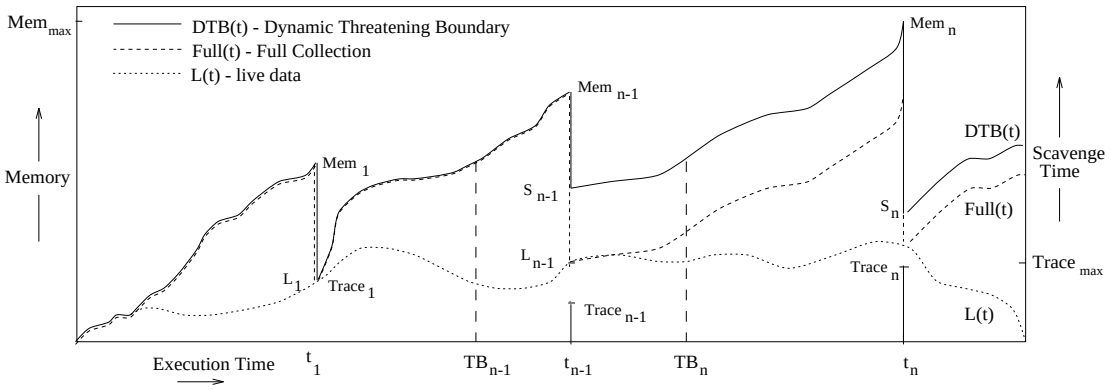

Figure 6.2: Garbage Collector Memory Use.
A non-generational full garbage collector collects all garbage at periodic intervals as shown by curve *Full* falling to curve L at time t_i, for $i = 1$ to n. like any generational collector, the dynamic threatening boundary mechanism saves tracing time by following curve DTB leaving some tenured garbage above the *Full* curve. DTB reduced tenured garbage after time t_n by selecting TB_n to trace older objects than TB_{n-1} did at time t_{n-1}. $Trace_{max}$ refers to a trace limit used by demographic policies such as feedback mediation and the dynamic threatening boundary.

data will survive the current collection by estimating that the survival rate for objects allocated since the last collection will be the same as from the previous collection. Therefore, it increments the sum obtained from the backward demographic scan by the number of newly-allocated bytes surviving during the last scavenge. Second, it attempts to avoid repetitive tracing of long-lived clumps of old objects (the so-called *pig-in-the-python*). When the policy exceeds $Trace_{max}$ as a result of moving the threatening boundary backward to a given value, it does not select that (or an older) boundary again at the next scavenge. Each unsuccessful attempt to collect the clump doubles the intervening time until the next attempt on that clump.

Figure 6.3 illustrates how the threatening boundary, TB_n, is selected at time t_n. Each of the birth-time regions $A - D$ contains objects still alive at time t_n that were born between collections at times t_i and t_{i-1} (for integer $i = 1$ to n). Because region E has never been collected, the first refinement uses D's value as a prediction. The new threatening boundary, TB_n, is set to the oldest value that does not exceed $Trace_{max} = 100$ thousand bytes (e.g., $E + D + C = 30 + 30 + 25 = 85 < 100$). TB_{n-x} illustrates a previous threatening boundary that was selected x collections ago as a result of the second refinement: collection of the long-lived region, A, was previously attempted, but resulted in too much storage being traced.

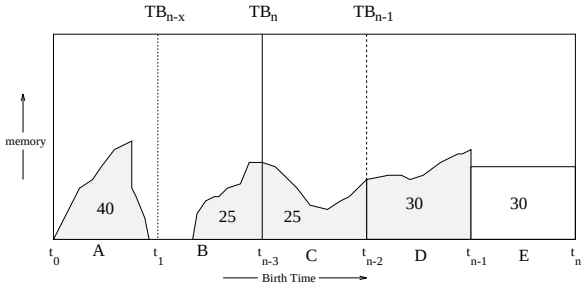

Figure 6.3: Threatening Boundary Demographics.
The numbers underneath each curve show the total storage traced by previous collections for each of the birth-time regions $A - D$. Each region contains objects born between collections (t_{i-1} to t_i, $i = 1$ to n) that are live at time t_n. Region E, to be traced, uses the previous collection's value, D, as an estimate. At time t_n, the DTB_{dg} collector moves threatening boundary backward from TB_{n-1} to TB_n by using the oldest boundary such that the total traced area is less than $Trace_{max} = 100$ thousand bytes ($E + D + C = 30 + 30 + 25 = 85 < 100$).

6.5 Methods

This section discusses the experimental technique and the assumptions made by my simulation model. Parameters for the collectors were selected to provide "reasonable" values which did not distort the results. Assumptions about implementation costs were chosen to be conservative by providing worst-case estimates; actual tuned implementations would likely yield better results.

I was interested in comparing the relative performance of different garbage collection algorithms in terms of CPU and memory overhead, tenured garbage, and bytes traced. To determine the effectiveness of the dynamic threatening boundary mechanism and the DTB_{dg} policy, I instrumented a set of five allocation-intensive C programs using ATOM [Dig93, SE94] as shown on the left-hand side of Figure 4.1 (p. 20). The programs measured are described in Table 4.1 (p. 21). I used memory allocation and deallocation events in these programs to drive a simulation of the different garbage collection algorithms. The output from the simulation consisted of memory and CPU usage patterns that were then processed to produce performance data.

The simulator implemented each of four garbage collection policies: full collection (FULL), fixed-age generational (FIXED1), feedback mediation (FM), and my collector policy (DTB_{dg}). Scavenges were triggered after every 1 million bytes of allocation and $Trace_{max}$ was set to 100 thousand bytes.

Conceptually, the new data allocated since the last collection is considered to be in the "nursery," although in a non-copying collector such data will not be contiguous. I assume that the nursery is always collected in all the algorithms I consider. Thus, what differentiates the threatening boundary selection policies I consider is what, in addition to the nursery, is collected.

I assumed a collection was triggered after every million bytes of allocation. As mentioned previously, the issue of when to collect exists for all collectors, not just generational ones. The collection interval chosen is often influenced by the program's allocation rate. Programs which have very high allocation rates execute very few instructions between each allocation and will be more sensitive to the collection interval. In equilibrium, such programs would be expected to have very high deallocation rates as well, which should make generational collection very effective. One million bytes was chosen as the collection interval to ensure that at least 20 garbage collections occurred for each application. Smaller collection intervals may cause collection overheads that are unrealistically high for a full-collection policy, or unfairly handicap a fixed-age collection policy by causing too much tenured garbage. The same fixed value was used to avoid introducing additional interactions which may distort the results.[1]

The trace limit ($Trace_{max}$) chosen affects the performance of the two demographic policies: FM and DTB_{dg}. The setting acts like a knob that increases pause times while reducing tenured garbage—or vice-versa. A high setting causes the collector to degenerate from a generational collector into a full collector. A low setting essentially disables garbage collection entirely and incurs maximum memory consumption and minimal collection overhead. I chose a value of 100 thousand bytes to prevent either form of degeneration to occur. As with the collection interval, the same value was chosen for all applications to avoid distorting the results.

To determine the CPU overhead added by DTB over existing generational collectors, I measured the size of the remembered sets and the frequency of forward-in-time stores for each application. I assumed that the remembered set size for DTB was at its maximum during the entire execution time of each program, and that it was zero for a generational collector. I assumed that each entry in the remembered set required an average of 20 instructions during the trace phase of each scavenge to perform the test for crossing the threatening boundary.[2] To determine the frequency of additions to the remembered set, I used the Forward Instr. column from Table 5.1 (p. 26) and assumed that each insertion required 20 additional instructions at each forward-in-time store.[3]

[1] One megabyte was also independently suggested based on measurements of collection in the ML language environment [SM94].

[2] Empty lists in a hash table can be detected in 4 instructions, the target birth-time in 12, and the source in 12 more. Since the threatened set is almost always smaller than the immune set, the target test will often fail, avoiding execution of the source test.

[3] 5 instructions for the hash function, 2 to index into the table of linked-lists, 6 to compare the key, and 7 for insertion.

To determine the memory required by the remembered set for the DTB mechanism, I measured the maximum number of live bytes, pointers, and forward-in-time pointers, for each application. I assumed that each remembered set entry required two pointer slots of 8 bytes each (as on the DEC Alpha), one for the address updated, and one for a hash link. Other tradeoffs may be made to improve memory or time performance as appropriate. For the purposes of comparison, I assumed other generational collectors had a remembered set size of zero, incurring no costs to update it.

To determine the memory overhead for maintaining the birth times, I measured the maximum number of live objects and multiplied by 8 bytes per object. To take into account memory fragmentation, the simulator allocated and deallocated dummy objects in a separate process using the GNU malloc/free routines as the simulation proceeded. I chose GNU malloc because it is among the most space-efficient allocation packages for C [DDZ94]. The maximum heap size was then recorded for each application.

6.6 Results

In this section, I investigate the costs of implementing the dynamic threatening boundary mechanism and the performance of the DTB_{dg} policy.

6.6.1 Cost of the DTB Mechanism

As I have mentioned, a dynamic threatening boundary mechanism is similar to that of a conventional generational collector, except that it must maintain a remembered set of *all* forward-in-time pointers. I investigate the overhead of creating and maintaining that set in this section.

CPU Overhead

In Figure 6.4, I show the cost of updating the remembered set (RS Insert), and scanning that set (RS Scan) during each collection. The numbers in the figure are derived from the measurements contained in Table 5.1 (p. 26). As the figure shows, the additional overhead associated with maintaining the write barrier (RS Insert) and scanning the larger remembered sets (RS Scan) programs ranges from 0% to 7% of total program execution time. The store check overhead for an in-line write barrier, as shown in Figure 5.1 (p. 27), will be the same for DTB as it is for any generational collector.

Memory Overhead

Tables 6.1 and 6.2 show the storage allocation behavior of each of the sample programs. The tables show the total bytes and objects allocated during the time the program ran, allocation rate, the maximum live objects, live bytes (excluding fragmentation), the maximum heap size (including fragmentation), number of objects, pointers, and remembered set size. The last three columns of Table 6.1 shows allocation rates for the C programs was ranged from 960 to 3,699 instructions between allocations. This information, combined with the total allocated bytes shows that a one megabyte collection interval met the criteria given on page 36.

Figure 6.5 shows the space overhead of maintaining all forward-in-time pointers in the remembered set and storing fine-grain birth times for each object as derived from Tables 6.1 and 6.2. Both the birth-field and the remembered-set overhead vary widely across the applications but range from 4% to 15% of total memory consumption. GAWK and CFRAC overheads are dominated by the birth-time field because they have mostly small objects and low pointer density. ESPRESSO and SIS have much higher pointer density, so remembered sets dominate their overhead.

Because I am modeling a non-copying collector, I conservatively assume that an 8-byte birth field is included with each object. In practice, only several hundred distinct birth-time values are probably necessary at any time, and so this information could be encoded in many fewer bits (e.g., by mapping object addresses to an array of bytes). Because FM also collects and uses object demographic information, it also requires such birth-time fields. In a copying collector, more efficient methods of encoding birth information are also possible. By co-locating objects of the same age on the same page (or sub-page), the 8-byte overhead would be incurred per page instead of per object. Such an approach could suffer memory loss from internal fragmentation, however.

Figure 6.5: Memory Overhead of the DTB Mechanism.
DTB incurs memory overhead for maintaining all forward-in-time pointers in the remembered set and a birth-time field for each object. The percentage of maximum memory consumed (including fragmentation) is shown. Note that feedback mediation would have the same birth-time field overhead.

Table 6.1: Program Allocation Behavior.

Program	Total Allocated		Maximum Live		Allocation Rate		
	KBytes	Objects	KBytes	Objects	instr. /alloc.	instr. /byte	bytes /alloc.
CFRAC	21,183	1,358,607	123	8,624	1,082	68	16
ESPRESSO	181,883	1,675,476	339	4,387	1,410	13	111
GAWK	235,027	1,802,220	4,720	119,403	960	7	134
GHOST	101,402	444,941	1,401	5,697	3,699	16	233
SIS	45,032	752,509	672	18,718	647	11	61

This table and Table 6.2 show the allocation behavior of the sample programs used for computing the memory overhead for the DTB mechanism. Assuming 8 bytes per object, the birth field overhead shown in Figure 6.5 is 8 times the Maximum Live Objects divided by Maximum Heap Size encountered during program execution (Table 6.2). For ESPRESSO, $1.80 = 100 \times (8 \times 4387)/(1900 \times 1024)$. The first two column of Allocation Rate are the ratio of Exec. Instr. (Table 5.1) to Total Allocated Objects and Total Allocated Bytes, respectively. The last column shows the average size per allocated object.

Table 6.2: Program Pointer Density Measurements.

Program	Maximum			% Forward Pointers	% Heap Pointers
	Heap Size KBytes	Rem. Set Pointers	Live Pointers		
CFRAC	1,536	1,189	3,149	38	20.04
ESPRESSO	1,900	9,403	21,451	43	49.44
GAWK	8,944	371	1,453	26	0.24
GHOST	3,684	5,562	35,993	15	20.07
SIS	2,928	18,291	35,884	51	41.73

Assuming 8-byte pointers, and two pointers for each remembered set element, the remembered set overhead shown in Figure 6.5 is 16 times Maximum Rem. Set divided by Maximum Heap Size. For ESPRESSO, $7.73 = 100 \times (16 \times 9403)/(1900 \times 1024)$. The pointer density (% Heap Pointers) is the ratio of 8 times Maximum Live Pointers to Maximum Live (Table 6.1). The forward-in-time pointer density (% Forward Pointers) is the ratio of remembered set pointers (Rem. Set Pointers) to Maximum Live Pointers. As the forward pointer density increases, so does the remembered set size and the corresponding overhead. Except for GHOST, the assumption that few pointers point forward-in-time appears unfounded for the C programs shown here.

6.6.2 Evaluation of DTB$_{dg}$ Policy

In this section, I evaluate the effectiveness of the DTB$_{dg}$ policy in reclaiming tenured garbage and compare it to three other collection algorithms. I first consider total memory use, then tenured garbage reclamation, its effect on pause times, and finally, the total amount of data traced. Note that I considered the fixed costs of maintaining the write barrier and scanning the remembered sets for the mechanism in the previous section; here I evaluate one specific policy that uses this mechanism.

Memory Benefits

I first looked at the memory required by each of the different collectors in Figures 6.6 and 6.7. In Figure 6.6, I evaluate collector performance with respect to maximum memory used (including fragmentation). DTB$_{dg}$ saved significant amounts of memory for ESPRESSO and GHOST even after the effects of the remembered set and birth time overheads from Figure 6.5 have been included. SIS still had a modest improvement even though it had the largest memory overhead due to its high pointer density. DTB$_{dg}$ incurs only a minor cost in CFRAC and GAWK where no collector distinguished itself.

Figure 6.7 shows how tenured garbage affected total memory use. CFRAC and GAWK do not generate significant amounts of tenured garbage. Thus there was no opportunity for either FM or DTB$_{dg}$ to recover much. For ESPRESSO and GHOST, however, tenured garbage was a significant portion of average memory use (50% and 30% respectively for FIXED1; 33% and 18% for FM). Observe how DTB$_{dg}$'s maximum memory use savings were roughly proportional to the proportion of tenured garbage. SIS had 22% tenured garbage for FM, but the savings were reduced by the 14% memory overhead of DTB (see Figure 6.5). When tenured garbage was present to a significant degree, my policy was able to collect much of it.

Object Tracing Costs

I now discuss the costs incurred as a result of collecting the tenured garbage. Figure 6.8 shows the 90th percentile pause times in the test programs (9 out of 10 pauses trace fewer bytes).[4] FULL is not shown because it had pause times off the chart. The values for FM and DTB$_{dg}$ are much lower than FULL, showing the benefits of generational garbage collection. FIXED1 had the lowest pause times, but at the cost of having the most tenured garbage (Figure 6.7). In SIS, the generational policies were not as effective at controlling pause times than they were in other programs, but they still reduced the 90th percentile pause-time of FULL from 654 to 225 thousand bytes (see Table 6.4). The figure shows what impact moving back the threatening boundary had on pause times. In particular, DTB$_{dg}$ was higher than FM and both were higher than FIXED1. As expected, the largest increase in pause times for DTB$_{dg}$ occurred for the same applications that had the greatest recovery of tenured garbage and was roughly proportional to the tenured garbage reclaimed.

Table 6.4: 90th Percentile Pause Times.

Program	(Ratio to $Trace_{max}$)			
	FULL	FIXED1	FM	DTB$_{dg}$
CFRAC	1.17	0.05	0.96	0.96
ESPRESSO	2.19	0.64	0.69	1.07
GAWK	43.62	0.21	1.01	1.19
GHOST	13.36	1.03	1.30	1.69
SIS	6.54	2.25	2.25	2.59

Each column shows the 90th percentile pause times for each program row as the ratio of total bytes traced to $Trace_{max}$ = 100 thousand bytes.

Data Traced

Finally, I consider the amount of work done (e.g., time spent) tracing data in the different collectors. Garbage collectors trade reduced memory consumption for added tracing CPU overhead. To convert bytes traced to CPU overhead, I assumed 20 instructions per byte-traced.[5] Consider that FULL would be completely off the scale in Figure 6.9, and compare the FIXED1 collector in Figure 6.9 with the FULL and FIXED1 collectors in Figure 6.6. FULL always traces all objects, and thus has the lowest memory consumption and the traces the most bytes. On the other hand, FIXED1 tenures objects after just one collection, and thus

[4] I used this metric to facilitate comparison with Ungar and Jackson's previous work [UJ92].

[5] Ungar and Jackson stated 8 microseconds per byte traced at 400 nanoseconds per instruction [UJ92, p. 10].

Figure 6.7: Mean Tenured Garbage.
The figure shows the mean amount of garbage that was tenured by each of the different collection algorithms during program execution for the FM and DTB_{dg} collectors. FULL has no tenured garbage by definition.

Figure 6.9: Total Bytes Traced.
This figure shows the cumulative total of all bytes traced by each collector during the lifetime of each program. FULL is not shown because it was off the scale in all cases (see Table 6.3).

Table 6.3: Total Data Traced.

Program	FM		(Ratio to FM)		
	KBytes	CPU(%)	FULL	FIXED1	DTB_{dg}
CFRAC	1,529	2.1	1.29	0.08	1.00
ESPRESSO	6,588	5.7	4.69	0.58	1.19
GAWK	1,721	25.7	26.64	0.23	1.00
GHOST	5,616	7.0	20.83	0.79	1.54
SIS	3,767	15.8	6.93	0.66	1.58

The two FM columns show the number of kilobytes traced and the CPU overhead for the FM collector. Subsequent rows show the relative performance of the other collectors as a ratio to FM= 1. The CPU Overhead was computed assuming 20 instructions per byte traced using Exec. Instr. from Table 5.1 (p. 26).

has the lowest tracing overhead but uses the most memory of the two. If tracing overhead was the sole concern of users, the FIXED1 policy would be the obvious choice because it has the lowest overhead. Unfortunately, this algorithm has the property that tenured garbage accumulates (e.g., quite rapidly in ESPRESSO and GHOST, see Figure 6.7) and its memory consumption is large, which motivates accepting the higher tracing overheads and pause-time overheads of FM and DTB_{dg}.

6.7 Summary

Using policy to define implementation motivates understanding of program behavior to derive effective policies to improve performance. Generational collection policies reduce pause times but cause excessive memory consumption by occasionally promoting objects that eventually become unreclaimed garbage. My DTB mechanism, unlike previous work, enables policies that can dynamically adjust the threatening boundary between immune and threatened data either forward or backward, essentially allowing objects to become untenured at any time. Measurements of five allocation-intensive C programs showed that the DTB mechanism added 4–15% to the total space and 0–7% to the total execution time of the programs over the costs of a conventional generational collection algorithm. I described one policy, DTB_{dg}, for setting the threatening boundary based on an extension of Ungar and Jackson's feedback mediation collector. My results showed the DTB_{dg} policy, which uses the dynamic threatening boundary mechanism, was more effective at reclaiming tenured garbage when it was present—in one case reducing memory requirements over feedback mediation by 50%.

Chapter 7

Lifetime Prediction

The previous chapter examined how policies can influence the performance of garbage collection; this chapter will show a couple of examples of how program behavior can influence policy selection. First, I will discuss how one insight into program behavior led to measurements that showed a high degree of correlation between short-lived objects and the point in the program responsible for allocating them. By a similar process, observations of program behavior during measurements of the performance of the write barrier and the dynamic threatening boundary collector led me to speculate about a possible correlation between stores and object lifetimes. The second section of this chapter shows some preliminary measurements supporting such a correlation. Unlike the previous chapters, this one will concentrate more upon behavior as a vehicle for suggesting future policies rather than on the precise mechanisms to implement those policies.

7.1 Motivation, Background and Scope

The principal feature of dynamic storage allocation is also its biggest liability. Dynamic storage allocation allows programs to create and use data structures whose memory consumption varies according to the demands of the program while it is running. The programmer need not decide in advance the worst-case memory consumption of the program. But, unlike static allocation, which can fail only when insufficient memory is available to load the program, dynamic allocation can run out of memory at unpredictable times. Virtual memory increases the address space available before memory exhaustion occurs, but the address space is not unlimited, especially on multi-programming systems.

One of the causes of memory exhaustion is fragmentation. As objects are allocated and deallocated, gaps are created in the address space when long-lived objects are assigned addresses between short-lived ones. If an allocation occurs, and and no gap is large enough to fit, more memory must be requested from the operating system, increasing the total memory required by the program. Fragmentation can account for a substantial fraction (22–85%) of the total memory consumed by a program [Wil95a]. Fragmentation occurs both for explicit dynamic storage allocation and for conservative garbage collection.

Garbage collectors can also suffer from excessive memory consumption. All garbage collectors defer deallocations until the trace phase completes. This delay ties up memory with unreachable objects for a time equal to the collection interval. Detlefs, Dosser and Zorn [DDZ94] measured a median overhead of 50% for garbage collection over explicit deallocation for eleven C and C++ applications. As mentioned in Chapter 6, generational collectors suffer memory loss from tenured garbage.

All these forms of memory loss depend highly upon program behavior. It has been proven (for systems that cannot move allocated objects) that pathological programs can always be devised that will run out of memory even though enough memory remains in the gaps to meet a given allocation request [Rob71]. Yet, most dynamic storage allocation systems perform well in practice; most programs behave in a manner that is not pathological. This chapter explores whether program behavior can be exploited further to reduce the peak memory overhead of dynamic storage allo-

cation systems, both for explicit deallocation and automatic garbage collection.

Generational collectors already exploit program behavior by concentrating effort upon objects likely to be short-lived. The heuristic they use to make this prediction is that the most recently allocated objects are the most likely to die. The measurements shown in this chapter will confirm this hypothesis, and suggest a possible extension.

Some of the measurements in this chapter are in units of time. The simplest measure is elapsed time, in seconds, between two events of interest. Typical important events are the start and end of program execution, or the allocation and deallocation times of an object. The age of an object is the difference between the current time, and the allocation time of the object.

Often, dynamic storage systems are only interested in time spent while doing storage management. For example, if an input–output-intensive program runs for several hours without doing any allocation or deallocation, this time is not important to a garbage collector or memory allocator since no storage allocation work is required. We would like a measurement that reflects only the time that the storage management system is operating.

The work load placed upon a storage allocator is dependent upon the rate of memory allocation requests. Often, the CPU cost is related to the object allocation rate, and the memory cost is related to the byte allocation rate. As a result, this chapter follows the convention that is common in storage allocation literature: time is measured in units of cumulative bytes allocated since program execution began. Thus, the lifetime of an object is the number of bytes that were allocated between the time the object was allocated and later freed.

7.2 Allocation-Site Lifetime Prediction

Profile-based optimization is often used as a technique to improve the CPU performance of a program. A representative execution of the program is instrumented to produce performance metrics that may be examined (either by a human being, or automatically by a compiler) and used to modify the program. The same technique may also be used to improve memory performance as well.

Programmers often have a sense of what portion of a program is responsible for allocating an object and when the allocation should take place. Frequently, a symmetric situation exists for object deallocations as well. Thus, object lifetimes should correlate with the location in the program responsible for object allocation.

The allocation site of an object corresponds to the point in the program responsible for allocating it. More precisely, the allocation site is the sequence of nodes on the path of the dynamic call graph from the allocation routine to the entry point of the program (e.g., the sequence of return addresses on the stack upon entry to `malloc`).

Hanson describes how segregating short-lived objects can improve memory performance, in his case by having the programmer explicitly specify what is short-lived [Han90]. If allocation sites do indeed correlate with object lifetimes, and profile-based optimization works, then it is possible to automate Hanson's algorithm.

Specifically, the approach would be to use an instrumentation tool, such as ATOM, to instrument the program to emit all the allocation sites with their corresponding object lifetime distributions. From the distributions, select those sites responsible for short-lived objects. Then, link in a special version of the allocation routine that tests to see if the allocation request is from one of the short-lived allocation sites. If so, then short-lived objects may be segregated into a special arena where they will not cause fragmentation by being interleaved in the rest of the address space with long-lived objects. A similar technique may be applied to conservative garbage collectors.

Tables 7.1 and 7.2 show how this idea would work by instrumenting a fictitious program to produce the allocation-site lifetime-distributions. The first table shows some of the allocation sites collected during one run of the program. Each allocation site corresponds to a call to `malloc`. A mapping function converts the sequence of nodes on the call graph for each site to an allocation site identifier. As each object is deallocated (or becomes unreachable), the lifetime distribution for the site identifier that allocated the object is updated. At the end of the program there a lifetime distribution corresponding to each site. The

Figure 7.1: Allocation Site Prediction Performance.
Of all the bytes allocated by the programs above, over 90% were short-lived. The allocation site was able to predict a high fraction of them. For three of the applications, all of the predicted short-lived bytes could be allocated to arenas (rightmost bar), thus avoiding fragmenting the heap.

7.3 Write-Barrier Lifetime Prediction

One of the problems that plagues generational garbage collectors is how to deal with long-lived clusters of objects. Chapter 6 showed how one policy for altering the threatening boundary reduced tenured garbage. As mentioned on page 35, a refine-

ment of the policy included a technique for recognizing the occurrence of a pig-in-the-python problem. Chapter 5 examined program store behavior relevant to an in-line write barrier. The barrier must track creation of forward-in-time pointers, but it can also track destruction of pointers as well.

The reachability of objects is reduced by pointer destructions, and when that reachablilty becomes zero, the object can be deallocated. Reference counting collection is simply a precise mechanism for ensuring this deallocation is performed promptly; perhaps a less precise mechanism may be useful for generational collection. The write barrier may be able assist with collecting long-lived immune objects by noting when pointers into the immune set are destroyed by non-initializing stores.

Hayes [Hay91] introduced the idea of *key objects* as policy for finding clusters of objects that are likely to be garbage. He noticed that at the time of deallocation, the majority of objects have the same allocation time (within one kilobyte). Key objects are those which, when they become unreachable, indicate a cluster of other unreachable objects. In later work Hayes [Hay93] presented an approach for clustering objects pointed to by key objects, and using the the targets of pointers in the remembered set as keys. Rather than using additions to the remembered set as key objects, I decided to investigate using deletions of pointers to immune objects instead.

In order to see if there was any merit to this hypothesis, I used the same techniques as in previous chapters with the same application programs. To the instrumentation code, I added a function that would output the object lifetime and store demographics (see the next section) after each 128 kilobytes of allocation. This demographic information was fed into a Matlab program to display the behavior graphically in three dimensions. Because of processor and memory limitations, I used shorter runs of the programs than with the measurements in previous chapters and not all of the programs were measured. I was looking for exploitable regularities in program behavior, and correlations between store events with object lifetimes in particular. Because the measurements were only for small amounts of allocation, the results presented are primarily to provide insight for future work.

Recall from Figure 6.3 (p. 35) that at a given instant in time, the live objects of a program were allocated at various different times. The demographic curve shows the number of currently live bytes that were allocated at each specific time. As program execution proceeds, the curve changes as new objects are allocated and old ones are freed. This measurement is similar to Stefanovic's [SM94] for ML but I use allocation-time demographics rather than age demographics.

Figure 7.2 shows the lifetime demographics for one of the application programs, ESPRESSO. Program execution began in the upper right corner and proceeded along the diagonal to the lower left. The vertical axis for the area to the right of the diagonal shows the number of bytes allocated at the time shown on the lower axis that still exist at the time shown on the left axis. The area behind the diagonal is zero because objects cannot exist before they are allocated.

ESPRESSO was fairly typical of the applications in several respects. First, notice the tall "fin" on the right. This represents data that was allocated at the time the program started, and lived almost the entire lifetime of the program. Notice also, that it is one of the largest sources of live data in the program. Such a fin was a common occurrence in the applications measured. Second, notice the spikes along the diagonal. These spikes represent strong confirmation of the generational hypothesis: large amounts of data die an extremely short time after allocation. Such behavior was also true of all the applications.

Further examination of the figure reveals more subtle behavior. Notice that several small lines extend from the diagonal into the foreground; two are at allocation times of about 7 and 9 kilobytes. These truncated fins ("pigs") represent small clusters of long-lived objects. Ideally, a generational collector would promote these objects shortly after allocation, and collect them shortly after they become unreachable. The dynamic threatening boundary collector would do this by moving the threatening boundary backward in time. For example, at 20 on the time axis, suppose a collection occurs. The collector could set the threatening boundary to 5 on the allocation-time axis. All unreachable objects consuming memory at time 20, which have allocation times more recent than (left of) 5 would be reclaimed. This would include the two long-lived clusters (pigs).

Notice that there are actually four very closely spaced fins on the right of the figure, a short one

on the very edge, one about 4 times taller next, followed by one half again as large, and finally by one half as tall that dies at about 6 kilobytes. A garbage collector trying to collect any but the last fin (which is actually a pig, because it died long before program termination) would be wasting its time. How does a collector recognize a fin, or more precisely, tell the difference between a fin, which is not reclaimable, and a pig, which is?

Figure 7.3 shows the store demographics for ESPRESSO. A spike on the figure corresponds to the number of stores that overwrite pointers to an object. The time axis corresponds to when the stores took place, and the allocation time axis corresponds to when the object pointed to by the overwriting store was allocated. Compare this store demographic figure with the lifetime demographic figure above. Observe that stores corresponding to long-lived objects are rare for this program. Also, notice there is a high correlation between destructive stores and the end of an object's lifetime as evinced by the store peaks in the foreground.

For a generational collector, this behavior could be exploited in a couple of ways: by triggering collections, or by adjusting the threatening boundary. All generational collectors must decide when to collect older generations. A sudden increase of stores overwriting pointers to immune objects could cause a flag to be set by the write barrier that would trigger a trace of older objects. For a dynamic threatening boundary collector, the demographics of the store could be used to select the threatening boundary explicitly.

Of course, not all applications can be counted upon to exhibit such polite behavior. For GHOST, shown in Figures 7.4 and 7.5, long-lived objects survived the entire execution of the program. Almost all destructive pointer-stores overwrote pointers to short-lived objects or to long-lived objects allocated at the start of program execution. Note that a collector using a store demographic policy would still work properly for GHOST if it ignored objects allocated near the start of program execution—it would correctly refuse to attempt to recover long-lived objects. CFRAC had similar behavior, except no stores changed pointers to long-lived objects.

This and the previous sections discussed how two observations of program behavior suggested

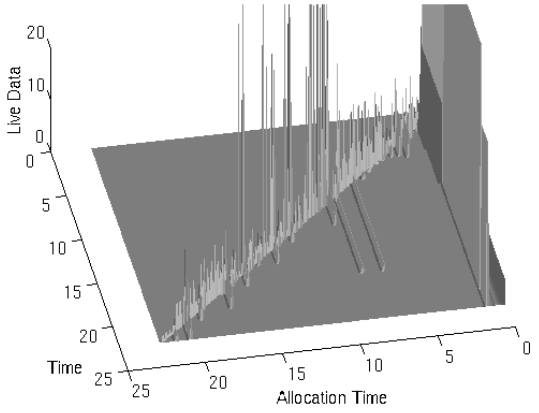

Figure 7.2: Lifetime Demographics for ESPRESSO.
This figure shows lifetime demographics as a function of time. As program execution proceeds from the back to the front of the figure, objects are allocated and later die. At any given instant, live data was allocated at various times, the oldest on the right, youngest on the left. The vertical axis shows the total live data. All units are in kilobytes (1024 bytes). The peak at the diagonal shows that most objects are short-lived.

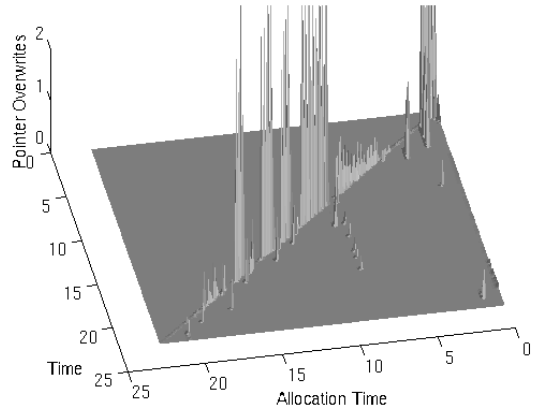

Figure 7.3: Pointer Overwrite Demographics for ESPRESSO.
Like Figure 7.2, this figure shows demographic information, but for stores that overwrite pointers. The vertical axis shows the number of stores that overwrite pointers to allocated objects as a function of time. Store peaks corresponded closely with lifetimes for this program.

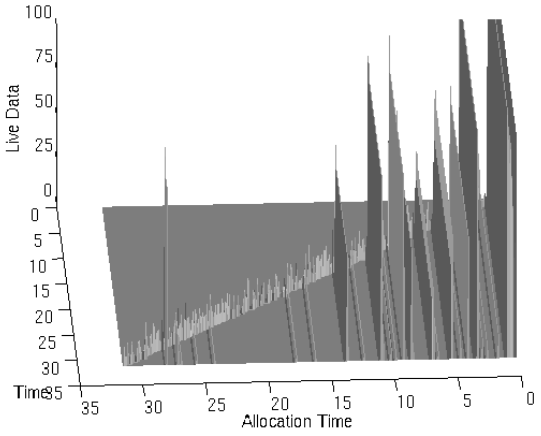

Figure 7.4: Lifetime Demographics for GHOST.
The lifetime demographics for GHOST were quite different than for ESPRESSO. Almost all long-lived objects survived until the program ended.

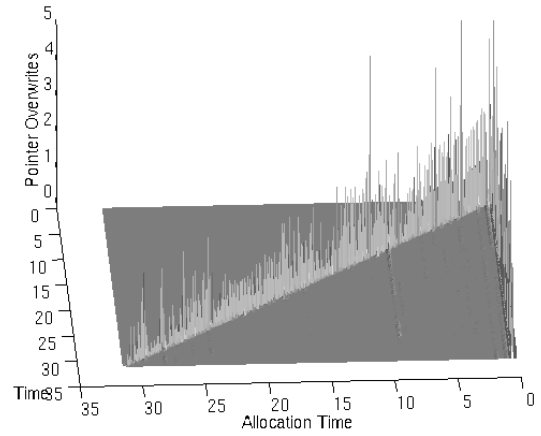

Figure 7.5: Pointer Overwrite Demographics for GHOST.
The destructive pointer store demographics for GHOST consisted almost entirely of pointers to objects allocated near the start of program execution.

policies to exploit that behavior: segregating objects based upon their allocation site, and using store behavior obtained by the write barrier to influence garbage collection collection scheduling or immune set selection. The allocation site provides an excellent predictor for short-lived objects and overwritten pointers show potential to predict the death of long-lived objects. In the latter case, only short runs were used, so further work is necessary before the idea can be validated. The techniques illustrated in this chapter provided insight into how powerful program instrumentation tools can be used to improve memory consumption in a manner similar to the way profiling tools have been used in the past to improve CPU performance.

This concludes the three chapters discussing each of the principal results of this dissertation. In the next chapter, I shall summarize the results obtained and how they might influence future work.

Chapter 8

Summary and Future Work

This dissertation has investigated three contributions for improving the performance of garbage collection: an in-line write barrier, a dynamic threatening boundary, and object lifetime prediction. Each of these contributions was investigated by using an instrumentation tool such as ATOM to directly alter the executable files for each of five different C programs. A simulator used events generated by the instrumented program to generate measurements of CPU and memory consumption. I made the additional contribution of collecting C program behaviors including store frequencies and storage allocation events, to provide a firm foundation for evaluating the performance of each of the other three contributions. This chapter summarizes these results, and relates them to each other and to future work.

A write barrier is used by generational collectors to detect pointers into the collected generation. The costs of this barrier for C programs has never been measured before. In this dissertation, I quantified the cost of using an explicit instruction sequence added at each store in five different allocation-intensive C programs. The CPU cost of an in-line write barrier was quite variable, ranging from 3% to almost 50% of total instructions executed. The pessimistic assumptions used by my model result in overheads that are probably too high for most of the C++ community.

However, there is ample opportunity for improvement. The write barrier need not check initializing or non-pointer stores. Technology that recognizes such stores can reduce the number of store instructions that must have the write-barrier check sequence added. Also, the number of instructions required by each store check may be reduced by using dedicated registers to facilitate rapid access to object ages and the remembered set.

Write barrier technology is also useful for incremental garbage collection, program debugging [RW93], persistent data structures (often used for object-oriented databases), and distributed data structures. Techniques that speed up the barrier increase its applicability to designers in each of these fields. The increased availability of program instrumentation tools, such as ATOM and EEL [LS95], make consideration of an in-line barrier attractive since there is no need to alter the compiler. Such instrumentation tools also make it possible to improve the performance of the write barrier both through code optimization [SW94] and exploiting program behavior through profiling.

Generational collectors use a threatening boundary to divide the young collected objects from the older uncollected ones. The policies for setting this threating boundary are often not explicit because the implementation defines the policy. In this dissertation I allow policies to be made explicit by introducing a a new mechanism, which extends generational collection to allows the threatening boundary to be adjusted at any collection to follow any desired policy. The dynamic threatening boundary mechanism provides a flexible tool for implementing generational garbage collectors in their most general form. I measured how one policy I designed, which extended Ungar and Jackson's feedback mediation, produced less than half as much tenured garbage of their already effective algorithm. The reduction in tenured garbage translated directly into improved peak memory consumption—reduced by as much as 50% for one application.

Further work exploring the relationship between program behavior and other threatening boundary selection policies appears promising. One possible technique, which avoids tracing objects allocated at the start of program execution and yet still captures long-lived objects, is likely to improve CPU overhead. Another (speculative) possibility is to set the threatening boundary while the trace proceeds rather than at the start of the trace. Traversing objects in birth-time order and terminating the trace at the point when a CPU trace threshold is reached would allow fine-tuning of the tradeoff between pause times and tenured garbage. [1]

In this dissertation, I showed that lifetime prediction using the allocation site is highly successful at finding short-lived objects, which comprise the majority (over 90%) of allocations made by the C programs measured. Store demographics obtained by the write barrier appears to have potential for predicting long-lived objects.

The allocation site lifetime predictor in this dissertation required a program execution profile to produce an explicit storage allocator that reduced memory consumption. An "on-the-fly" approach, which collected and exploited behavior while the program was running, would eliminate the need for obtaining a representative profile. Lifetime predictors that use a policy designed to improve reference locality are also possible. Either technique could improve both explicit storage deallocation and conservative garbage collection.

In this dissertation, I showed measurements of object lifetimes and store behavior that support using overwritten pointers in the heap to drive collection policies. I presented a hypothesis where pointers that are overwritten indicate which objects have a higher probability of dying and when they are more likely to die. One possibility for exploiting this behavior is to trigger collections based upon bursts of store activity. Another is to set the threatening boundary to the ages of objects pointed to by overwritten pointers. A non-generational policy could select key objects, which point to clusters of objects more likely to be dead, based upon the addresses of overwritten pointers. These techniques correspond to an imprecise form of reference-counting garbage collection.

[1] This technique is analogous to terminating search trees in computer games (e.g., chess) based upon a time constraint.

Program behavior guided the selection of policies and provided the basis for evaluation of results throughout this dissertation. C program behavior relevant for garbage collection has not been measured before. Interestingly, unlike for LISP or ML, pointers observed here did not confirm the belief that pointers from old to young objects were rare; 15–51% of pointers in the heap were forward-in-time.

The generational hypothesis was confirmed: recently allocated objects are disproportionally more likely to die than older objects. However, long-lived objects were frequently allocated during execution, and their presence complicates generational policies.

The programs measured here performed a significant amount of memory allocation, and had a wide variety of behaviors. These behaviors indicate the need to study more applications, especially those written in C++ and that have larger maximum memory sizes. More precise information about the number of initializing stores is also needed. Behaviors necessary for quantifying exploitable behavior of long-lived objects would help derive policies for reducing fragmentation, generation selection, and scheduling of collections. From those policies, implementations would follow.

The technology now exists to easily produce fully-implemented versions of the ideas simulated for this dissertation. The Boehm conservative collector is available as a library that can be compiled to replace the `malloc` (*new*) library in existing C and C++ programs [Boe95a]. A straight-forward version of a DTB mechanism can be implemented by adding the object allocation-time to the object descriptor data structure. An in-line write barrier can be added by using tools such as ATOM to instrument all the stores in an program's executable file. Lifetime prediction can be implemented by using an instrumentation tool to collect profiles and to alter the executable or produce a more efficient allocation package in the form of a library.

As collector implementations become more widely available for C and C++, and their performance is improved, programmers will be more likely to take advantage of that technology to reduce the time to produce new products and increase their reliability. Garbage collection provides one tool that can help meet these objectives.

Glossary

age For *allocated objects*, the *time* since it became allocated. For events, the time since the event occured.

allocated object An *object* whose storage was obtained from a dynamic allocation routine (e.g., malloc, new, cons), and is not eligible for reallocation.

allocation-site The point in the control-flow-graph of an executing program where a new object is allocated from primary memory. One representation of the allocation-site is the sequence of source-file/line-number pairs appearing on the subroutine call *stack* at the *time* of a memory allocation event.

allocation time The *time* when an object became an *allocated object*. Contrast with *age*.

ambiguous pointer A value that appears to be a valid *pointer*. It may actually be a pointer, but it may also be a *false* or *derived* pointer instead.

birth time A synonym for *allocation time*.

collection Invocation of a *garbage collector* to attempt to identify and deallocate *garbage*. Short for garbage collection.

collection interval The *time* between invocation of each *collection*. The collection interval is chosen according to some policy. Typical policies are when a memory limit is exceeded or after a specified amount of allocation has taken place since the last collection.

collector Short for *garbage collector*.

conservative collector A *garbage collector* that assumes that all values that look like *pointers* (according to a *pointer-finding heuristic*), are indeed pointers. They must not break a program if the assumption is incorrect (such as by altering the value of a pointer when an object is copied). Conservative collectors do not require language or computer architecture support to identify pointers.

copying collector A *garbage collector* that copies *reachable objects* into *tospace* as they are discovered during the *trace phase*.

card marking An implementation for a *remembered set* where each element of the set corresponds to a sequence of addresses (called *cards*) to *scan* for pointers rather than a single pointer address. Each address contains a potential pointer from a *immune object* to a *threatened object*. Cards are typically much smaller than virtual memory pages to reduce unnecessary scanning.

dangling reference A *pointer* from a *reachable object* to a *deallocated object*.

dead object Synonym for *garbage*.

deallocated object An *allocated object* whose storage has been made eligible for reallocation for another object. Objects may be deallocated explicitly by the programmer calling a deallocation routine (e.g., free, delete), or implicitly by a *garbage collector*. Once deallocated, the object ceases to exist.

deallocation time The *time* when an object became a *deallocated object*.

deferred sweep For a *mark-sweep collector*, delaying the *sweep phase* until subsequent allocations occur. Each allocation performs part of the sweep by checking a portion of the mark bits.

derived pointer A *pointer* whose target is determined from a computation rather than from a single stored value. A value that is a pointer,

but does not correspond to the lowest address of an allocated object.

dynamic threatening boundary For *generational collectors*, a *threatening boundary* that may be changed from one *collection* to the next.

external fragmentation The space consumed by *deallocated objects* occupying multiple contiguous regions of storage.

false pointer A value that is not a pointer, but corresponds to an address of an allocated object.

feedback mediation A type of *generational garbage collection* where objects are *promoted* only when a specified trace limit (number of bytes traced) is exceeded during the *trace phase*.

fixed-age policy A *threatening boundary* selection policy that sets the threatening boundary to the current *time* at each *collection* minus some fixed interval of time (typically the time of the previous collection).

forward-in-time pointer A *pointer* from an older *object* to a younger one.

forwarding pointer A *pointer* that is stored into the *fromspace* copy of an *object* immediately after that object has been copied into *tospace* during the *trace phase* of a *copying collector*. This pointer indicates that the object has already been copied and the new address of the object.

full collection A garbage collection policy in which all *allocated objects* are in the *threatened set* at every *collection*. See also *fixed-age*, *feedback mediation*, and *dynamic threatening boundary*.

fromspace During the *trace phase* of a *copying collector*, the region of the address space where *objects* are being copied from. When the trace phase terminates, the entire fromspace is freed and becomes available for allocation of new objects.

garbage An allocated *object* that is no longer accessible to the program and is thus eligible for deallocation without causing erroneous behavior. A more restricted definition includes objects that are accessible, but will never be accessed again.

garbage collector An algorithm that identifies *garbage* and reclaims its storage by deallocating it. Garbage is identified as all *objects* that are allocated but not *reachable*. Reachable objects are identified as objects transitively reachable from *pointers* in *roots*. Identifying reachable objects is called *tracing*.

generation A set of allocated *objects* sharing the same allocation time class. Typically objects allocated after a certain time before the current collection are placed into an old generation and those before into a young generation.

generational collector A *garbage collector* that attempts to reclaim recently allocated objects more often than older objects by segregating them into *generations*.

heap For this dissertation, the region of the memory address space used for storing dynamically *allocated objects*. Objects in the heap are never *roots*. The heap size may be expanded during program execution by calls to the operating system.

immune set During a *collection* those *objects* that are ineligible for deallocation. Often, a collector may save time by not *tracing* these objects and using a *write barrier* to maintain a *remembered set* of *pointers* from the *immune set* into the *threatened set*.

inter-generational pointer A *pointer* from an *object* in one *generation* to another. Almost always, they are also *forward-in-time* pointers.

inter-generational store An instruction that stores an *inter-generational pointer*.

interior pointer A *pointer* into an *object* rather than to the lowest address of the object.

internal fragmentation The space wasted by a dynamic storage system by reserving more storage for each *allocated object* than was requested.

lifetime The interval of *time* between when an *object* becomes *allocated* and *deallocated.*

live object An allocated object that is transitively *reachable* from the *roots.*

mark phase The *trace phase* of a *mark-sweep collector*; *objects* are explicitly marked as *reachable* by setting a corresponding mark bit.

mark-sweep collector A *garbage collector* that explicitly marks all *reachable objects* and then makes a sweep explicitly deallocating all unmarked ones. A set of mark bits is maintained, one for each allocated object. Upon initiating a collection, the mark bits are cleared. Starting from the *roots*, all reachable objects are visited and their corresponding mark bit is set during the *mark phase.* Afterwards, each un-marked object is *deallocated* during the *sweep phase.*

memory leak *Garbage* that will never be *deallocated.*

mutator The program that is allocating, updating, and reading the storage being reclaimed by a *garbage collector.*

nepotism For a *generational collector*, the phenomenon of *garbage* in the *threatened set* that is ineligible for reclamation because it is being directly pointed to by garbage in the *immune set.*

newspace For a *copying collector*, the region of the address space where *objects* are initially *allocated.*

object A region of sequential random-access memory addresses (locations) and their values dedicated to a specific purpose. If the memory becomes used for a new purpose, such as by deallocation and subsequent reallocation, the new object is distinct from the older one.

pig-in-the-python A cluster of *objects* allocated at about the same *time* that becomes *unreachable* at about the same time, and lives significantly longer than objects allocated during its own *lifetime.* Pigs increase the cost of *generational collectors* either by creating *tenured garbage* if they remain in the *immune set* or by causing excessive CPU consumption during the *trace phase* if they remain in the *threatened set.*

pointer A random-access memory address corresponding to a given *object* (called the *target*) appearing as a value within the same or a different object (called the *source*). The pointer is said to point from the source object to the target object. The source object *references* the target object.

pointer-finding heuristic For a *conservative collector*, a predicate that tests a value to see if it is possibly a *pointer* to an *allocated object.* The test must be conservative; it must never return false for a valid pointer, but it may return true for an invalid one—at the cost of unnecessarily retaining memory.

promotion For a *generational collector*, the act of reclassifying an *object* from the *threatened set* to the *immune set.*

reachable object An *allocated object* that is *referenced* directly by a *root* or by another *reachable object.*

reference Synonym for *pointer.*

reference counting A method to avoid erroneously reclaiming an *shared object* whereby a counter is maintained for each object recording the number of *references* to that object. The reference count is initialized to zero when the object is allocated, incremented when a new reference to it is created, and decremented when a reference is destroyed. The object is *deallocated* when the reference count is decremented to zero.

remembered set The set of *rescuers* corresponding to a specific *immune set.* During the *trace* phase of a *generational collector*, the remembered set for the generation being *collected* augments the *root set.*

rescuer A *pointer* from an *object* in the *immune set* to (or into) an object in the *threatened set.* Such pointers are maintained by a *write barrier* that updates a data structure called the *remembered-set.*

root An *object* always directly accessible to the program. Typical examples of roots are processor registers, statically allocated objects, and the program *stack*.

root set The set of all *roots*.

scanning The operation of examining one or more *objects* for pointers. Scanning an object is cheaper than *tracing* because it is a more primitive operation.

scavenge Synonym for *collection*.

shared object An *object* that is transitively *reachable* via more than one path from the *roots*. Also, an object that is reachable from an entity outside the current program (such an an I/O device, or another processor). The latter definition is outside the scope of this dissertation.

stack The region of memory managed by the compiler to store variables whose *lifetime* and scope match the invocation of a subroutine or function call. *Pointers* to variables in the stack are forbidden to be returned by the subroutine.

sweep phase After a *mark-sweep collector* completes identifying *reachable objects* during the *mark phase*, the process of examining all the mark bits, and explicitly deallocating all unmarked objects.

tenured garbage *Garbage* that will not be reclaimed by a *generational collector* because of *promotion* into a *generation* for old *objects* (the *immune set*), which is not traced by the collector.

tenuring Same as *promotion*.

threatened set During a *collection* those *objects* that will be discovered to be either *live* or *garbage* when the collection completes. Threatened objects are eligible for deallocation if they are found to be *unreachable*. For *generational collectors*, the objects in the younger generations are threatened, and older generations are *immune*.

threatening boundary In a *generational collector*, the boundary that divides the *threatened set* from the *immune set*. The threatening boundary corresponds to a specific instant in *time* where objects allocated before that instant are threatened and those after are immune.

time I can not find a definition of time that is not circular. Stephen Hawking probably understands what time is as well as anyone [Haw89]. For the purposes of this dissertation, time can me measured in seconds, instructions, cycles, or in total bytes allocated since the allocating program started execution.

tospace During the *trace phase* of a *copying collector*, the region of the address space where *objects* surviving the current *collection* are copied into from *fromspace*.

trace phase The act of identifying the currently *reachable* objects in the *threatened set*. Reachable objects are identified as objects transitively reachable from *rescuers* or *pointers* in *roots*.

tracing Collector operation during the *trace phase*. Tracing an *object* consists of *scanning* it for pointers, and then recursively *tracing* each of the objects pointed to by those pointers.

unreachable object An *allocated object* that is not a *reachable object*.

write barrier After a collection, *unreachable threatened objects* may still not be *garbage*, because of *rescuers*. Rescuers are discovered during program execution when store instructions occur. Each store is checked (either by a page-protection trap or an explicit instruction sequence) to see if it creates a rescuer; if so, the rescuer is added to the *remembered set*. The write barrier consists of all store checks for this purpose.

Bibliography

[AEL88] Andrew Appel, John Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 11–20, Atlanta, GA, June 1988. SIGPLAN, ACM Press.

[App92] Apple Computer Inc. *Macintosh Common Lisp Reference*, version 2 edition, 1992. pages 631–637.

[Bak78] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[Bak93] Henry G. Baker, Jr. Infant mortality and generational garbage collection. *ACM SIGPLAN Notices*, 28(4):55–77, April 1993.

[Bar90] Joel Bartlett. A generational, compacting garbage collector for C++. In *OOPSLA'90 Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, CA, October 1990.

[BDS91] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *ACM SIGPLAN SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 157–164, June 1991. Toronto, Ontario, Canada.

[BGS94] David F. Bacon, Susan L. Graham, and Olivar J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[Boe91] Hans-J. Boehm. Simple GC-safe compilation. In *The 1991 OOPSLA Workshop on Garbage Collection in Object Oriented Systems*, October 1991.

[Boe93] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 197–205, Albuquerque, New Mexico, June 1993.

[Boe94] Hans-Juergen Boehm. Garbage collector implementation. Available online via anonymous FTP from *parcftp.xerox.com:/pub/gc/* or by the URL *ftp://ftp.parc.xerox.com /pub/gc/gc.html*, April 1994. Version 4.0.

[Boe95a] Hans-Juergen Boehm. Garbage collector implementation. See [Boe94], July 1995. File *os_dep.c* under "Routines for accessing dirty bits on virtual pages", version 4.4.

[Boe95b] Hans-Juergen Boehm. Personal Communication. At ACM PLDI'95 Conference, June 1995.

[BW88] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, pages 807–820, September 1988.

[BZ93] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 187–196, Albuquerque, New Mexico, June 1993.

[BZ95] David A. Barrett and Benjamin G. Zorn. Garbage collection using a dynamic threatening boundary. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 301–314, La Jolla, California, June 1995.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[CL89] E.G. Coffman, Jr. and F. T. Leighton. A provably efficient algorithm for dynamic storage allocation. *Journal of Computer and System Sciences*, 38(1):2–35, February 1989.

[Coh81] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.

[Col60] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 2(12):655–657, December 1960.

[Cou88] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9):1128–1138, September 1988.

[CWB86] Patrick Caudill and Allen Wirfs-Brock. A third generation Smalltalk-80 implementation. In Normam Meyrowitz, editor, *OOPSLA '86 Conference Proceedings*, pages 119–130, Portland, OR, September 1986. ACM.

[DDZ94] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. *Software Practice and Experience*, 24(6):527–542, June 1994.

[Det93] Dave Detlefs. Empirical evidence for using garbage collection in C and C++ programs. In *OOPSLA '93 Workshop on Memory Management and Garbage Collection*, Washington, D.C., September 1993.

[Dig91] Digital Equipment Corporation. *Unix Manual Page for Pixie*, ULTRIX V4.2 (rev 96) edition, September 1991.

[Dig93] Digital Equipment Corporation, Maynard, Massachusetts. *ATOM Reference Manual*, December 1993.

[DMH92] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically-typed language. In *Proceedings of the 1992 SIGPLAN Conference on Programming Language Design and Implementation*, pages 273–282, San Francisco, California, June 1992.

[DWH+90] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Conference Record of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 261–269, San Francisco, California, January 1990.

[ED94] John R. Ellis and David L. Detlefs. Safe, efficient garbage collection for C++. In *Proceedings of the 1994 USENIX C++ Conference*, pages 143–177, Cambridge, Massachusetts, April 1994. also as Technical Report TR-102, Digital Equipment Corporation, Palo Also, California, June 1993.

[Fra92] Franz Inc. *Allegro CL User Guide, Version 4.1*, revision 2 edition, March 1992. Chapter 15: Garbage Collection.

[FY69] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[GZ93] Dirk Grunwald and Benjamin Zorn. CustoMalloc: Efficient synthesized memory allocators. *Software Practice and Experience*, 23(8):851–869, August 1993.

[GZH93] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 177–186, Albuquerque, June 1993.

[Ḧ93] Urs Hölzle. A fast write barrier for generational garbage collectors. In *OOPSLA'93 Workshop on Memory Management and Garbage Collection*, Washington, D.C., September 1993.

[Han77] David R. Hanson. Storage management for an implementation of SNOBOL4. *Software Practice and Experience*, 7:179–192, 1977.

[Han90] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software Practice and Experience*, 20(1):5–12, January 1990.

[Haw89] Stephen W. Hawking. *A Brief History of Time : From the Big Bang to Black Holes*. Bantam Books, New York, New York, 1989.

[Hay90] Barry Hayes. A theoretical basis for garbage collection. Draft copy, Nov 1990.

[Hay91] Barry Hayes. Using key object opportunism to collect old objects. In *ACM SIGPLAN 1991 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '91)*, pages 33–46, Phoenix, Arizona, October 1991. ACM Press.

[Hay93] Barry Hayes. Finding key objects in cross-generational stores. In *OOPSLA'93 Workshop on Memory Management and Garbage Collection*, Washington, D.C., September 1993.

[HH93] Antony L. Hosking and Richard L. Hudson. Remembered sets can also play cards. In *OOPSLA'93 Workshop on Memory Management and Garbage Collection*, Washington, D.C., September 1993.

[HJ92] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–136, San Francisco, California, January 1992. USENIX Association. January 20–24.

[HM92] Richard L. Hudson and J. Eliot B. Moss. Incremental collection of mature objects. In *Proceedings of the International Workshop on Memory Management*, pages 388–403, St. Malo, France, September 1992. Springer-Verlag Lecture Notes in Computer Science vol. 637.

[HM93] Antony L. Hosking and J. Eliot B. Moss. Protection traps and alternatives for memory management of an object-oriented language. In *Proceedings of the faurteenth ACM Symposium on Operating Systems Principles*, pages 106–119, Asheville, NC, December 1993.

[HMD91] R. L. Hudson, J. E. B. Moss, and A. Diwan. A language independent garbage collector toolkit. Technical Report COINS TR 91-47, University of Massachusetts, Amherst, MA, September 1991.

[HMS92] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *ACM SIGPLAN 1992 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '92)*, pages 92–109, Vancouver, British Columbia, Canada, October 1992.

[Knu73] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*, chapter 2, pages 435–451. Addison Wesley, Reading, MA, 2nd edition, 1973.

[KV85] David G. Korn and Kiem-Phong Vo. In search of a better malloc. In *Proceedings of the Summer 1985 USENIX Conference*, pages 489–506, 1985.

[LB94] Jamer R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software Practice and Experience*, 24(2):197–218, February 1994. previously Technical Report 1083, Computer Sciences Department, University of Wisconsin–Madison, 1210 West Dayton Street, Madison, WI 53706, USA, March, 1992.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[LS95] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 291–300, La Jolla, California, June 1995.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computations by machine, part I. *Communications of the ACM*, 3(4):184–195, April 1960.

[MK88] Marshall Kirk McKusick and Michael J. Karels. Design of a general purpose memory allocator for the 4.3BSD UNIX kernel. In *Summer USENIX '88 Conference Proceedings*, pages 295–303, San Francisco, California, June 1988.

[Moo84] David A. Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 235–246, Austin, Texas, August 1984.

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3.* Series in Innovative Technology. Prentice Hall, Reading, MA, 1991. ISBN 0-13-590464-1, L.C. QA76.66.S87, $34.40.

[NO93] Scott Nettles and James O'Toole. Real-time replication garbage collection. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 217–226, Albuquerque, New Mexico, June 1993.

[Ran69] B. Randell. A note on storage fragmentation and program segmentation. *Communications of the ACM*, 12(7):365–369,372, July 1969.

[Rob71] J. M. Robson. An estimate of the store size necessary for dynamic storage allocation. *Journal of the Association for Computing Machinery*, 18(3):416–423, July 1971.

[RW93] Susan L. Graham Robert Wahbe, Steven Lucco. Practical data breakpoints: Design and implementation. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 1–12, Albuquerque, New Mexico, June 1993.

[SE94] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, Orlando, FL, June 1994.

[Sha87] Robert A. Shaw. Improving garbage collector performance in virtual memory. Technical Report CSL-TR-87-323, Stanford University, March 1987.

[Sha88] Robert A. Shaw. *Empirical Analysis of a Lisp System.* PhD thesis, Stanford University, Stanford, California, February 1988. Also appears as Technical Report CSL-TR-88-351, Stanford University Computer Systems Laboratory, 1988.

[SM94] Darko Stefanović and J. Eliot B. Moss. Characterisation of object behavior in standard ML of new jersey. In *Conference Record of the 1994*

ACM Symposium on LISP and Functional Programming, pages 43–54, Orlando, Florida, June 1994.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the Association for Computing Machinery*, 32(3):652–686, July 1985.

[Sta80] Thomas Standish. *Data Structures Techniques*. Addison-Wesley Publishing Company, 1980.

[SW67] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[SW94] Amitabh Srivastava and David W. Wall. Link-time optimization of address calculation on a 64-bit architecture. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 49–64, Orlando, FL, June 1994.

[Sym85] Symbolics Inc. *Symbolics LISP Manual: Internals, Processes, and Storage Management #996085*, release 6.0 edition, March 1985. Pages 113–130.

[TL94] C. A. Thekkath and H. M. Levy. Hardware and software support for efficient exception handling. *ACM SIGPLAN Notices*, 29(11):110–119, November 1994.

[UJ92] David Ungar and Frank Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, January 1992.

[Ung84] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *SIGSOFT/SIGPLAN Practical Programming Environments Conference*, pages 157–167, April 1984.

[Wal72] David C. Walden. A note on Cheney's nonrecursive list-compacting algorithm. *Communications of the ACM*, 15(4):275, April 1972.

[WDBH94] Mark D. Weiser, Alan J. Demers, Daniel G. Bobrow, and Barry Hayes. Method and system for reclaiming unreferenced computer memory space. United States Patent No. 5,321,834, June 1994.

[Wen90] E. P. Wentworth. Pitfalls of conservative garbage collection. *Software Practice and Experience*, 20(7):719–727, July 1990.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, pages 1–42, St. Malo, France, September 1992. Springer-Verlag Lecture Notes in Computer Science vol. 637.

[Wil95a] Paul R. Wilson. Dynamic storage allocation: A survey and review. In *Proceedings of the International Workshop on Memory Management*, Scotland, 1995. Springer-Verlag Lecture Notes in Computer Science, to appear,.

[Wil95b] Paul R. Wilson. Uniprocessor garbage collection tenchniques. *ACM Computing Surveys*, to appear 1995. Previously appeared as [Wil92].

[WM89a] Paul R. Wilson and Thomas G. Moher. A 'card-marking' scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices*, 24(5):87–92, May 1989.

[WM89b] Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In *ACM SIGPLAN 1989 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '89)*, pages 23–35, New Orleans, Louisiana, October 1989.

[WW88] C.B. Weinstock and W. A. Wulf. Quickfit: An efficient algorithm for heap storage allocation. *ACM SIGPLAN Notices*, 23(10):141–144, October 1988.

[Xer83] Xerox Corporation. *Interlisp Reference Manual*, October 1983. pages 18.20-21,22.7-11.

[ZG92] Benjamin Zorn and Dirk Grunwald. Empirical measurements of six allocation-intensive C programs. *ACM SIGPLAN Notices*, 27(12):71–80, December 1992.

[ZG94] Benjamin Zorn and Dirk Grunwald. Evaluating models of memory allocation. *ACM Transactions on Modeling and Computer Simulation*, 4(1):107–131, January 1994.

[Zor89] Benjamin Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, Berkeley, California, November 1989. Also appears as tech report UCB/CSD 89/544.

[Zor90a] Benjamin Zorn. Barrier methods for garbage collection. Technical Report CU-CS-494-90, Department of Computer Science, University of Colorado, Boulder, CO, November 1990.

[Zor90b] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *1990 ACM Conference on Lisp and Functional Programming*, pages 87–98, Nice, France, June 1990.

Index